
boost_histogram

Henry Schreiner, Hans Dembinski

Apr 24, 2024

USER GUIDE:

1	Installation	3
2	Quickstart	5
3	Histogram	11
4	Axes	15
5	Storages	19
6	Accumulators	21
7	Using Transforms	25
8	Indexing	29
9	Plotting	31
10	Analyses examples	33
11	NumPy compatibility	35
12	Subclassing (advanced)	37
13	Comparison with Boost.Histogram	39
14	Simple Example	41
15	ROOT file format example	43
16	Threaded Fills	45
17	Performance Comparison	47
18	XArray Example	51
19	Using boost-histogram	59
20	Contributing	71
21	Support	77
22	Changelog	79

23	boost_histogram	95
24	boost_histogram.axis	99
25	boost_histogram.axis.transform	103
26	boost_histogram.accumulators	105
27	boost_histogram.numpy	107
28	boost_histogram.storage	113
29	boost_histogram.tag	115
30	boost_histogram.version	117
31	Acknowledgements	119
32	Indices and tables	121
	Python Module Index	123
	Index	125



Boost-histogram ([source](#)) is a Python package providing Python bindings for `Boost.Histogram` ([source](#)). You can install this library from [PyPI](#) with `pip` or you can use Conda via [conda-forge](#):

```
python -m pip install boost-histogram
```

```
conda install -c conda-forge boost-histogram
```

All the normal best-practices for Python apply; you should be in a virtual environment, etc. See [Installation](#) for more details. An example of usage:

```
import boost_histogram as bh

# Compose axis however you like; this is a 2D histogram
hist = bh.Histogram(bh.axis.Regular(2, 0, 1), bh.axis.Regular(4, 0.0, 1.0))

# Filling can be done with arrays, one per dimension
hist.fill([0.3, 0.5, 0.2], [0.1, 0.4, 0.9])

# NumPy array view into histogram counts, no overflow bins
counts = hist.view()
```

See [Quickstart](#) for more.

INSTALLATION

Boost-histogram ([source](#)) is a Python package providing Python bindings for [Boost.Histogram](#) ([source](#)).

You can install this library from [PyPI](#) with pip:

```
python -m pip install boost-histogram
```

or you can use Conda through [conda-forge](#):

```
conda install -c conda-forge boost-histogram
```

All the normal best-practices for Python apply; you should be in a virtual environment, etc.

1.1 Supported platforms

1.1.1 Binaries available:

The supported platforms are listed in the README - All common linux machines, all common macOS versions, and all common Windows versions.

1.1.2 Conda-Forge

The boost-histogram package is available on Conda-Forge, as well. All supported versions are available.

```
conda install -c conda-forge boost-histogram
```

1.1.3 Source builds

For a source build, for example from an “sdist” package, the only requirements are a C++14 compatible compiler. The compiler requirements are dictated by Boost.Histogram’s C++ requirements: gcc >= 5.5, clang >= 3.8, msvc >= 14.1. You should have a version of pip less than 2-3 years old (10+).

NumPy is downloaded during the build (enables multithreaded builds). Boost is not required or needed (this only depends on included header-only dependencies). This library is under active development; you can install directly from GitHub if you would like.

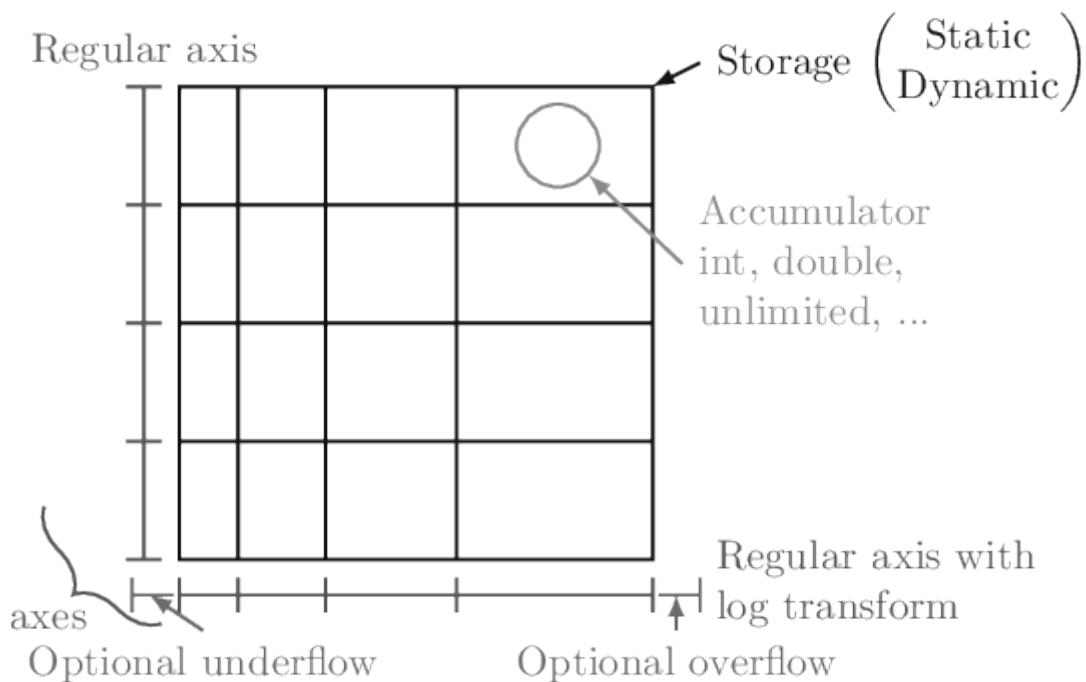
```
python -m pip install git+https://github.com/scikit-hep/boost-histogram.git@develop
```


QUICKSTART

All of the examples will assume the following import:

```
import boost_histogram as bh
```

In boost-histogram, a histogram is collection of Axis objects and a storage.



2.1 Making a histogram

You can make a histogram like this:

```
hist = bh.Histogram(bh.axis.Regular(bins=10, start=0, stop=1))
```

If you'd like to type less, you can leave out the keywords:

```
hist = bh.Histogram(bh.axis.Regular(10, 0, 1))
```

The exact same syntax is used for 1D, 2D, and ND histograms:

```
hist3D = bh.Histogram(
    bh.axis.Regular(10, 0, 100, circular=True),
    bh.axis.Regular(10, 0.0, 10.0),
    bh.axis.Variable([1, 2, 3, 4, 5, 5.5, 6]),
)
```

See [Axes](#) and [Using Transforms](#).

You can also select a different storage with the `storage=` keyword argument; see [Storages](#) for details about the other storages.

2.2 Filling a histogram

Once you have a histogram, you can fill it using `.fill`. Ideally, you should give arrays, but single values work as well:

```
hist = bh.Histogram(bh.axis.Regular(10, 0.0, 1.0))
hist.fill(0.9)
hist.fill([0.9, 0.3, 0.4])
```

2.3 Slicing and rebinning

You can slice into a histogram using bin coordinates or data coordinates using `bh.loc(v)`. You can also rebin with `bh.rebin(n)` or remove an entire axis using `sum` (technically as the third slice argument, though it is allowed by itself as well):

```
hist = bh.Histogram(
    bh.axis.Regular(10, 0, 1),
    bh.axis.Regular(10, 0, 1),
    bh.axis.Regular(10, 0, 1),
)
mini = hist[1:5, bh.loc(0.2) : bh.loc(0.9), sum]
# Will be 4 bins x 7 bins
```

See [Indexing](#).

2.4 Accessing the contents

You can use `hist.values()` to get a NumPy array from any histogram. You can get the variances with `hist.variances()`, though if you fill an unweighted storage with weights, this will return `None`, as you no longer can compute the variances correctly (please use a weighted storage if you need to). You can also get the number of entries in a bin with `.counts()`; this will return counts even if your storage is a mean storage. See [Plotting](#).

If you want access to the full underlying storage, `.view()` will return a NumPy array for simple storages or a `RecArray`-like wrapper for non-simple storages. Most methods offer an optional keyword argument that you can pass, `flow=True`, to enable the under and overflow bins (disabled by default).

```
np_array = hist.view()
```

2.5 Setting the contents

You can set the contents directly as you would a NumPy array; you can set either values or arrays at a time:

```
hist[2] = 3.5
hist[bh.underflow] = 0 # set the underflow bin
hist2d[3:5, 2:4] = np.eye(2) # set with array
```

For non-simple storages, you can add an extra dimension that matches the constructor arguments of that accumulator. For example, if you want to fill a Weight histogram with three values, you can dimension:

```
hist[0:3] = [[1, 0.1], [2, 0.2], [3, 0.3]]
```

See [Indexing](#).

2.6 Accessing Axes

The axes are directly available in the histogram, and you can access a variety of properties, such as the `edges` or the `centers`. All properties and methods are also available directly on the `axes` tuple:

```
ax0 = hist.axes[0]
X, Y = hist.axes.centers
```

See [Axes](#).

2.7 Saving Histograms

You can save histograms using pickle:

```
import pickle

with open("file.pkl", "wb") as f:
    pickle.dump(h, f)

with open("file.pkl", "rb") as f:
    h2 = pickle.load(f)

assert h == h2
```

Special care was taken to ensure that this is fast and efficient. Please use the latest version of the Pickle protocol you feel comfortable using; you cannot use version 0, the version that was default on Python 2. The most recent versions provide performance benefits.

2.8 Computing with Histograms

As an complete example, let's say you wanted to compute and plot the density:

```
#!/usr/bin/env python3

from __future__ import annotations

import functools
import operator

import matplotlib.pyplot as plt
import numpy as np

import boost_histogram as bh

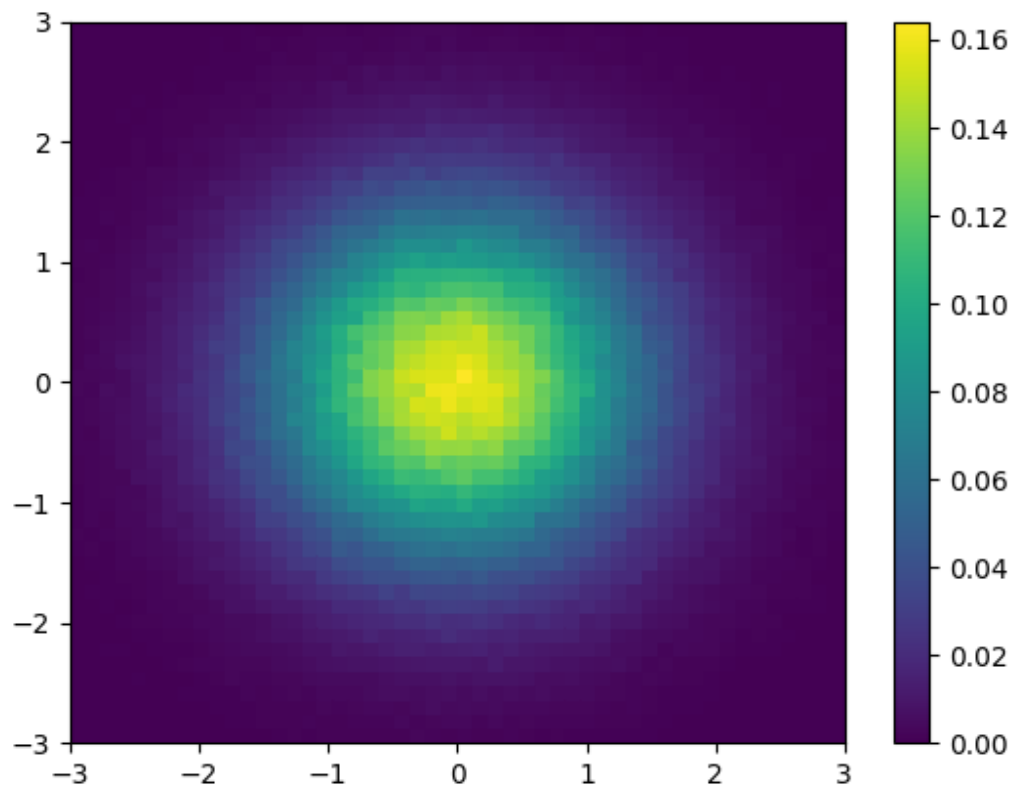
# Make a 2D histogram
hist = bh.Histogram(bh.axis.Regular(50, -3, 3), bh.axis.Regular(50, -3, 3))

# Fill with Gaussian random values
hist.fill(np.random.normal(size=1_000_000), np.random.normal(size=1_000_000))

# Compute the areas of each bin
areas = functools.reduce(operator.mul, hist.axes.widths)

# Compute the density
density = hist.view() / hist.sum() / areas

# Make the plot
fig, ax = plt.subplots()
mesh = ax.pcolormesh(*hist.axes.edges.T, density.T)
fig.colorbar(mesh)
plt.savefig("simple_density.png")
```



2.9 Comparing with Boost.Histogram

This is built on the Boost.Histogram library.

See *Comparison with Boost.Histogram*.

HISTOGRAM

The Histogram object is the core of boost-histogram.

3.1 Filling

You call `.fill` to fill. You must have one 1D array (or scalar value) per dimension. For maximum performance, numeric arrays should be continuously laid out in memory, and either 64-bit floats or ints. If any other layouts or numeric datatypes are supplied, a temporary copy will be made internally before filling.

All storages support a `weight=` parameter, and some storages support a `sample=` parameter. If supplied, they must be a scalar (applies to all items equally) or an iterable of scalars/1D arrays that matches the number of dimensions of the histogram.

The summing accumulators (not `Mean()` and `WeightedMean()`) support threaded filling. Pass `threads=N` to the fill parameter to fill with N threads (and using 0 will select the number of virtual cores on your system). This is helpful only if you have a large number of entries compared to your number of bins, as all non-atomic storages will make copies for each thread, and then will recombine after the fill is complete.

3.2 Data

The primary values from a histogram are always available as `.values()`. The variances are available as `.variances()`, unless you fill an unweighed histogram with weights, which will cause this to return `None`, since the variances are no longer computable (use a weighted storage instead if you need the variances). The counts are available as `.counts()`. If the histogram is weighted, `.counts()` returns the effective counts; see [UHI](#) for details.

3.3 Views

While Histograms do conform to the Python buffer protocol, the best way to get access to the raw contents of a histogram as a NumPy array is with `.view()`. This way you can optionally pass `flow=True` to get the flow bins, and if you have an accumulator storage, you will get a View, which is a slightly augmented ndarray subclass (see [Accumulators](#)). Views support setting as well for non-computed properties; you can use an expression like this to set the values of an accumulator storage:

```
h.view().value = values
```

You can also used stacked arrays (N+1 dimensional) to set a histogram's contents. This is especially useful if you need to set a computed value, like variance on a Mean/WeightedMean storage, which cannot be set using the above method:

```
h[...] = np.stack([values, variances], axis=-1)
```

If you leave endpoints off (such as with ... above), then you can match the size with or without flow bins.

3.4 Operations

- `h.rank`: The number of dimensions
- `h.size` or `len(h)`: The number of bins
- `+`: Add two histograms, or add a scalar or array (storages must match types currently)
- `*=`: Multiply by a scalar, array, or histogram (not all storages) (`hist * scalar` and `scalar * hist` supported too)
- `/=`: Divide by a scalar, array, or histogram (not all storages) (`hist / scalar` supported too)
- `[...]`: Access a bin or a range of bins (get or set) (see [Indexing](#))
- `.sum(flow=False)`: The total count of all bins
- `.project(ax1, ax2, ...)`: Project down to listed axis (numbers)
- `.to_numpy(flow=False, view=False)`: Convert to a NumPy style tuple (with or without under/overflow bins, and either return values (the default) or the entire view for accumulator storages.)
- `.view(flow=False)`: Get a view on the bin contents (with or without under/overflow bins)
- `.values(flow=False)`: Get a view on the values (counts or means, depending on storage)
- `.variances(flow=False)`: Get the variances if available
- `.counts(flow=False)`: Get the effective counts for all storage types
- `.reset()`: Set counters to 0
- `.empty(flow=False)`: Check to see if the histogram is empty (can check flow bins too if asked)
- `.copy(deep=False)`: Make a copy of a histogram
- `.axes`: Get the axes as a tuple-like (all properties of axes are available too)
 - `.axes[0]`: Get the 0th axis
 - `.axes.edges`: The lower values as a broadcasting-ready array
 - `.axes.centers`: The centers of the bins broadcasting-ready array
 - `.axes.widths`: The bin widths as a broadcasting-ready array
 - `.axes.metadata`: A tuple of the axes metadata
 - `.axes.traits`: A tuple of the axes traits
 - `.axes.size`: A tuple of the axes sizes (size without flow)
 - `.axes.extent`: A tuple of the axes extents (size with flow)
 - `.axes.bin(*args)`: Returns the bin edges as a tuple of pairs (continuous axis) or values (describe)
 - `.axes.index(*args)`: Returns the bin index at a value for each axis
 - `.axes.value(*args)`: Returns the bin value at an index for each axis

3.5 Saving a Histogram

You can save a histogram using pickle:

```
import pickle

with open("file.pkl", "wb") as f:
    pickle.dump(h, f)

with open("file.pkl", "rb") as f:
    h2 = pickle.load(f)

assert h == h2
```

Special care was taken to ensure that this is fast and efficient. Please use the latest version of the Pickle protocol you feel comfortable using; you cannot use version 0, the version that used to be default on Python 2. The most recent versions provide performance benefits.

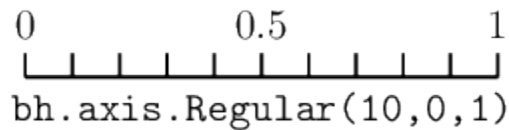
You can nest this in other Python structures, like dictionaries, and save those instead.

In boost-histogram, a histogram is collection of Axis objects and a storage.

4.1 Axis types

There are several axis types to choose from.

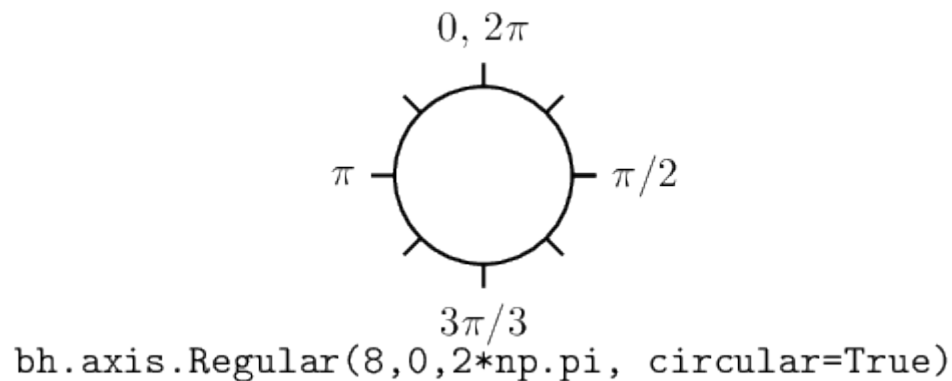
4.1.1 Regular axis



`bh.axis.Regular(bins, start, stop, *, metadata="", underflow=True, overflow=True, circular=False, growth=False, transform=None)`

The regular axis can have overflow and/or underflow bins (enabled by default). It can also grow if `growth=True` is given. In general, you should not mix options, as growing axis will already have the correct flow bin settings. The exception is `underflow=False, overflow=False`, which is quite useful together to make an axis with no flow bins at all.

There are some other useful axis types based on regular axis:



`bh.axis.Regular(..., circular=True)`

This wraps around, so that out-of-range values map back into the valid range circularly.

4.1.2 Regular axis: Transforms

Regular axes support transforms, as well; these are functions that convert from an external, non-regular bin spacing to an internal, regularly spaced one. A transform is made of two functions, a **forward** function, which converts external to internal (and for which the transform is usually named), and a **inverse** function, which converts from the internal space back to the external space. If you know the functional form of your spacing, you can get the benefits of a constant performance scaling just like you would with a normal regular axis, rather than falling back to a variable axis and a poorer scaling from the bin edge lookup required there.

You can define your own functions for transforms, see [Using Transforms](#). If you use compiled/numba functions, you can keep the high performance you would expect from a Regular axis. There are also several precompiled transforms:

```
bh.axis.Regular(..., transform=bh.axis.transform.sqrt)
```

This is an axis with bins transformed by a sqrt.

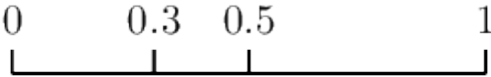
```
bh.axis.Regular(..., transform=bh.axis.transform.log)
```

Transformed by log.

```
bh.axis.Regular(..., transform=bh.axis.transform.Power(v))
```

Transformed by a power (the argument is the power).

4.1.3 Variable axis



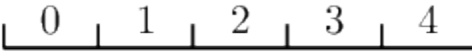
A horizontal axis line with four tick marks. Above the first tick mark is the number '0', above the second is '0.3', above the third is '0.5', and above the fourth is '1'.

```
bh.axis.Variable([0, .3, .5, 1])
```

```
bh.axis.Variable([edge1, ..., ], *, metadata="", underflow=True, overflow=True, circular=False, growth=False)
```

You can set the bin edges explicitly with a variable axis. The options are mostly the same as the Regular axis.

4.1.4 Integer axis



A horizontal axis line with five tick marks. Above each tick mark is a number: '0', '1', '2', '3', and '4' from left to right.

```
bh.axis.Integer(0, 5)
```

```
bh.axis.Integer(start, stop, *, metadata="", underflow=True, overflow=True, circular=False, growth=False)
```

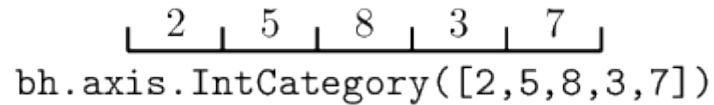
This could be mimicked with a regular axis, but is simpler and slightly faster. Bins are whole integers only, so there is no need to specify the number of bins.

One common use for an integer axis could be a true/false axis:

```
bool_axis = bh.axis.Integer(0, 2, underflow=False, overflow=False)
```

Another could be for an IntEnum if the values are contiguous.

4.2 Category axis



```

bh.axis.IntCategory([2,5,8,3,7])

```

```

bh.axis.IntCategory([value1, ..., ]metadata="", growth=False)

```

You should put integers in a category axis; but unlike an integer axis, the integers do not need to be adjacent.

One use for an IntCategory axis is for an IntEnum:

```

import enum

class MyEnum(enum.IntEnum):
    a = 1
    b = 5

my_enum_axis = bh.axis.IntEnum(list(MyEnum), underflow=False, overflow=False)

```

```

bh.axis.StrCategory([str1, ..., ]metadata="", growth=False)

```

You can put strings in a category axis as well. The fill method supports lists or arrays of strings to allow this to be filled.

4.3 Manipulating Axes

Axes have a variety of methods and properties that are useful. When inside a histogram, you can also access these directly on the `hist.axes` object, and they return a tuple of valid results. If the property or method normally returns an array, the `axes` version returns a broadcasting-ready version in the output tuple.

STORAGES

There are several storages to choose from. To select a storage, pass the `storage=bh.storage.` argument when making a histogram.

5.1 Simple storages

These storages hold a single value that keeps track of a count, possibly a weighed count.

5.1.1 Double

By default, boost-histogram selects the `Double()` storage. For most uses, this should be ideal. It is just as fast as the `Int64()` storage, it can fill up to 53 bits of information (9 quadrillion) counts per cell, and supports weighted fills. It can also be scaled by a floating point values without making a copy.

```
h = bh.Histogram(bh.axis.Regular(10, 0, 1)) # Double() is the default
h.fill([0.2, 0.3], weight=[0.5, 2]) # Weights are optional
print(f"{h[bh.loc(.2)]=}\\n{h[bh.loc(.3)]=}") # Python 3.8 print
```

```
h[bh.loc(.2)]=0.5
h[bh.loc(.3)]=2.0
```

5.1.2 Unlimited

The Unlimited storage starts as an 8-bit integer and grows, and converts to a double if weights are used (or, currently, if a view is requested). This allows you to keep the memory usage minimal, at the expense of occasionally making an internal copy.

5.1.3 Int64

A true integer storage is provided, as well; this storage has the `np.uint64` datatype. This eventually should provide type safety by not accepting non-integer fills for data that should represent raw, unweighed counts.

```
h = bh.Histogram(bh.axis.Regular(10, 0, 1), storage=bh.storage.Int64())
h.fill([0.2, 0.3], weight=[1, 2]) # Integer weights supported
print(f"{h[bh.loc(.2)]=}\\n{h[bh.loc(.2)]=}")
```

```
h[bh.loc(.2)]=1  
h[bh.loc(.3)]=2
```

5.1.4 AtomicInt64

This storage is like `Int64()`, but also provides a thread safety guarantee. You can fill a single histogram from multiple threads.

5.2 Accumulator storages

These storages hold more than one number internally. They return a smart view when queried with `.view()`; see *Accumulators* for information on each accumulator and view.

5.2.1 Weight

This storage keeps a sum of weights as well (in CERN ROOT, this is like calling `.Sumw2()` before filling a histogram). It uses the `WeightedSum` accumulator.

5.2.2 Mean

This storage tracks a “Profile”, that is, the mean value of the accumulation instead of the sum. It stores the count (as a double), the mean, and a term that is used to compute the variance. When filling, you must add a `sample=` term.

5.2.3 WeightedMean

This is similar to `Mean`, but also keeps track a sum of weights like term as well.

ACCUMULATORS

6.1 Common properties

All accumulators can be filled like a histogram. You just call `.fill` with values, and this looks and behaves like filling a single-bin or “scalar” histogram. Like histograms, the fill is inplace.

All accumulators have a `.value` property as well, which gives the primary value being accumulated.

6.2 Types

There are several accumulators.

6.2.1 Sum

This is the simplest accumulator, and is never returned from a histogram. This is internally used by the Double and Unlimited storages to perform sums when needed. It uses a highly accurate Neumaier sum to compute the floating point sum with a correction term. Since this accumulator is never returned by a histogram, it is not available in a view form, but only as a single accumulator for comparison and access to the algorithm. Usage example in Python 3.8, showing how non-accurate sums fail to produce the obvious answer, 2.0:

```
import math
import numpy as np
import boost_histogram as bh

values = [1.0, 1e100, 1.0, -1e100]
print(f"{sum(values) = } (simple)")
print(f"{math.fsum(values) = }")
print(f"{np.sum(values) = } (pairwise)")
print(f"{bh.accumulators.Sum().fill(values) = }")
```

```
sum(values) = 0.0 (simple)
math.fsum(values) = 2.0
np.sum(values) = 0.0 (pairwise)
bh.accumulators.Sum().fill(values) = Sum(0 + 2)
```

Note that this is still intended for performance and does not guarantee correctness as `math.fsum` does. In general, you must not have more than two orders of values:

```
values = [1., 1e100, 1e50, 1., -1e50, -1e100]
print(f"{math.fsum(values) = }")
print(f"{bh.accumulators.Sum().fill(values) = }")
```

```
math.fsum(values) = 2.0
bh.accumulators.Sum().fill(values) = Sum(0 + 0)
```

You should note that this is a highly contrived example and the Sum accumulator should still outperform simple and pairwise summation methods for a minimal performance cost. Most notably, you have to have large cancellations with negative values, which histograms generally do not have.

You can use += with a float value or a Sum to fill as well.

6.2.2 WeightedSum

This accumulator is contained in the Weight storage, and supports Views. It provides two values; .value, and .variance. The value is the sum of the weights, and the variance is the sum of the squared weights.

For example, you could sum the following values:

```
import boost_histogram as bh

values = [10]*10
smooth = bh.accumulators.WeightedSum().fill(values)
print(f"{smooth = }")

values = [1]*9 + [91]
rough = bh.accumulators.WeightedSum().fill(values)
print(f"{rough = }")
```

```
smooth = WeightedSum(value=100, variance=1000)
rough = WeightedSum(value=100, variance=8290)
```

When filling, you can optionally provide a variance= keyword, with either a single value or a matching length array of values.

You can also fill with += on a value or another WeightedSum.

6.2.3 Mean

This accumulator is contained in the Mean storage, and supports Views. It provides three values; .count, .value, and .variance. Internally, the variance is stored as _sum_of_deltas_squared, which is used to compute variance.

For example, you could compute the mean of the following values:

```
import boost_histogram as bh

values = [10]*10
smooth = bh.accumulators.Mean().fill(values)
print(f"{smooth = }")

values = [1]*9 + [91]
```

(continues on next page)

(continued from previous page)

```
rough = bh.accumulators.Mean().fill(values)
print(f"{rough = }")
```

```
smooth = Mean(count=10, value=10, variance=0)
rough = Mean(count=10, value=10, variance=810)
```

You can add a `weight=` keyword when filling, with either a single value or a matching length array of values.

You can call a `Mean` with a value or with another `Mean` to fill inplace, as well.

6.2.4 WeightedMean

This accumulator is contained in the `WeightedMean` storage, and supports Views. It provides four values; `.sum_of_weights`, `.sum_of_weights_squared`, `.value`, and `.variance`. Internally, the variance is stored as `._sum_of_weighted_deltas_squared`, which is used to compute variance.

For example, you could compute the mean of the following values:

```
import boost_histogram as bh

values = [1]*9 + [91]
wm = bh.accumulators.WeightedMean().fill(values, weight=2)
print(f"{wm = }")
```

```
wm = WeightedMean(sum_of_weights=20, sum_of_weights_squared=40, value=10, variance=810)
```

You can add a `weight=` keyword when filling, with either a single value or a matching length array of values.

You can call a `WeightedMean` with a value or with another `WeightedMean` to fill inplace, as well.

6.3 Views

Most of the accumulators (except `Sum`) support a `View`. This is what is returned from a histogram when `.view()` is requested. This is a structured NumPy ndarray, with a few small additions to make them easier to work with. Like a NumPy recarray, you can access the fields with attributes; you can even access (but not set) computed attributes like `.variance`. A view will also return an accumulator instance if you select a single item. You can set a view's contents with a stacked array, and each item in the stack will be used for the (computed) values that a normal constructor would take. For example, `WeightedMean` can take an array with a final dimension four long, with `sum_of_weights`, `sum_of_weights_squared`, `value`, and `variance` elements, even though several of these values are computed from the internal representation.

USING TRANSFORMS

The boost-histogram library provides a powerful transform system on Regular axes that allows you to provide a functional form for the conversion between a regular spacing and the actual bin edges. The following transforms are built in:

- `bh.axis.transform.sqrt`: A square root transform
- `bh.axis.transform.log`: A logarithmic transform
- `bh.axis.transform.Pow(power)` Raise to a specified power (`power=0.5` is identical to `sqrt`)

There is also a flexible `bh.axis.transform.Function`, which allows you to specify arbitrary conversion functions (detailed below).

7.1 Simple custom transforms

The `Function` transform takes two ctypes `double(double)` function pointers, a forward transform and a inverse transform. An object that provides a ctypes function pointer through a `.ctypes` attribute is supported, as well. As an example, let's look at how one would recreate the `log` transform using several different methods:

7.1.1 Pure Python

You can directly cast a python callable to a ctypes pointer, and use that. However, you will call Python *every* time you interact with the transformed axis, and this will be 15-90 times slower than a compiled method, like `bh.axis.transform.log`. In most cases, a `Variable` axis will be faster.

```
import ctypes

ftype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)

# Pure Python (15x slower)
bh.axis.Regular(
    10, 1, 4, transform=bh.axis.transform.Function(ftype(math.log), ftype(math.exp))
)

# Pure Python: NumPy (90x slower)
bh.axis.Regular(
    10, 1, 4, transform=bh.axis.transform.Function(ftype(np.log), ftype(np.exp))
)
```

You can create a `Variable` axis from the edges of this axis; often that will be faster.

You can also use `transform=ftype` and just directly provide the functions; this provides nicer reprs, but is still not picklable because `ftype` is a generated and not picklable; see below for a way to make this picklable. You can also specify `name="..."` to customize the repr explicitly.

7.1.2 Using Numba

If you have the numba library installed, and your transform is reasonably simple, you can use the `@numba.cfunc` decorator to create a callable that will run directly through the C interface. This is just as fast as the compiled version provided!

```
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)

bh.axis.Regular(10, 1, 4, transform=bh.axis.transform.Function(log, exp))
```

7.1.3 Manual compilation

You can also get a ctypes pointer from the usual place: a library. Let's say you have the following `mylib.c` file:

```
#include <math.h>

double my_log(double value) {
    return log(value);
}

double my_exp(double value) {
    return exp(value);
}
```

And you compile it with:

```
gcc mylib.c -shared -o mylib.so
```

You can now use it like this:

```
import ctypes

ftype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)

mylib = ctypes.CDLL("mylib.so")

my_log = ctypes.cast(mylib.my_log, ftype)
```

(continues on next page)

(continued from previous page)

```
my_exp = ctypes.cast(mylib.my_exp, ftype)

bh.axis.Regular(10, 1, 4, transform=bh.axis.transform.Function(my_log, my_exp))
```

Note that you do actually have to cast it to the correct function type; just setting `argtypes` and `restype` does not work.

7.2 Picklable custom transforms

The above examples do not support pickling, since `ctypes` pointers (or pointers in general) are not picklable. However, the `Function` transform supports a `convert=` keyword argument that takes the two provided objects and converts them to `ctypes` pointers. So if you can supply a pair of picklable objects and a conversion function, you can make a fully picklable transform. A few common cases are given below.

7.2.1 Pure Python

This is the easiest example; as long as your Python function is picklable, all you need to do is move the `ctypes` call into the `convert` function. You need a little wrapper function to make it picklable:

```
import ctypes, math

# We need a little wrapper function only because ftype is not directly picklable
def convert_python(func):
    ftype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
    return ftype(func)

bh.axis.Regular(
    10,
    1,
    4,
    transform=bh.axis.transform.Function(math.log, math.exp, convert=convert_python),
)
```

That's it.

7.2.2 Using Numba

The same procedure works for `numba` decorators. `NumPy` only supports functions, not builtins like `math.log`, so if you want to pass those, you'll need to wrap them in a `lambda` function or add a bit of logic to the `convert` function. Here are your options:

```
import numba, math

def convert_numba(func):
    return numba.cfunc(numba.double(numba.double))(func)
```

(continues on next page)

(continued from previous page)

```
# Built-ins and ufuncs need to be wrapped (numba can't read a signature)
# User functions would not need the lambda
bh.axis.Regular(
    10,
    1,
    4,
    transform=bh.axis.transform.Function(
        lambda x: math.log(x), lambda x: math.exp(x), convert=convert_numba
    ),
)
```

Note that `numba.cfunc` does not work on its own builtins, but requires a user function. Since with the exception of the simple example I'm showing here that is already available directly in `boost-histogram`, you will probably be composing your own functions out of more than one builtin operation, you generally will not need the `lambda` here.

7.2.3 Manual compilation

You can use strings to look up functions in the shared library:

```
def lookup(name):
    mylib = ctypes.CDLL("mylib.so")
    function = getattr(mylib, name)
    return ctypes.cast(function, ftype)

bh.axis.Regular(
    10, 1, 4, transform=bh.axis.transform.Function("my_log", "my_exp", convert=lookup)
)
```


INDEXING

Boost-histogram implements the UHI indexing protocol. You can read more about it on the [UHI Indexing](#) page.

8.1 Boost-histogram specific details

Boost-histogram implements `bh.loc`, `builtins.sum`, `bh.rebin`, `bh.underflow`, and `bh.overflow` from the UHI spec. A `bh.tag.at` locator is provided as well, which simulates the Boost.Histogram C++ `.at()` indexing using the UHI locator protocol.

Boost-histogram allows “picking” using lists, similar to NumPy. If you select with multiple lists, boost-histogram instead selects per-axis, rather than group-selecting and reducing to a single axis, like NumPy does. You can use `bh.loc(...)` inside these lists.

Example:

```
h = bh.histogram(
    bh.axis.Regular(10, 0, 1),
    bh.axis.StrCategory(["a", "b", "c"]),
    bh.axis.IntCategory([5, 6, 7]),
)

minihist = h[:, [bh.loc("a"), bh.loc("c")], [0, 2]]

# Produces a 3D histogram with Regular(10, 0, 1) x StrCategory(["a", "c"]) x
↳ IntCategory([5, 7])
```

This feature is considered experimental in boost-histogram 1.1.0. Removed bins are not added to the overflow bin currently.

PLOTTING

Boost-histogram does not contain plotting functions - this is outside of the scope of the project, which is histogram filling and manipulation. However, it does follow `PlottableProtocol`. Any plotting library that accepts an object that follows the `PlottableProtocol` can plot boost-histogram objects.

Read about the `PlottableProtocol` in the [UHI plotting](#) page.

ANALYSES EXAMPLES

10.1 Bool and category axes

Taken together, the flexibility in axes and the tools to easily sum over axes can be applied to transform the way you approach analysis with histograms. For example, let's say you are presented with the following data in a 3xN table:

Data	Details
value	
is_valid	True or False
run_number	A collection of integers

In a traditional analysis, you might bin over value where is_valid is True, and then make a collection of histograms, one for each run number. With boost-histogram, you can make a single histogram, and use an axis for each:

```
value_ax = bh.axis.Regular(100, -5, 5)
bool_ax = bh.axis.Integer(0, 2, underflow=False, overflow=False)
run_number_ax = bh.axis.IntCategory([], growth=True)
```

Now, you can use these axes to create a single histogram that you can fill. If you want to get a histogram of all run numbers and just the True is_valid selection, you can use a sum:

```
h1 = hist[:, True, sum]
```

You can expand this example to any number of dimensions, boolean flags, and categories.

NUMPY COMPATIBILITY

11.1 Histogram conversion

11.1.1 Accessing the storage array

You can access the storage of any Histogram using `.view()`, see *Histogram*.

11.1.2 NumPy tuple output

You can directly convert a histogram into the tuple of outputs that `np.histogram*` would give you using `.to_numpy()` or `.to_numpy(flow=True)` on any histogram. This returns `edges[0]`, `edges[1]`, ..., `values`, and the edges are NumPy-style (upper edge inclusive).

11.2 NumPy adaptors

You can use boost-histogram as a drop in replacement for NumPy histograms. All three histogram functions (`bh.numpy.histogram`, `bh.numpy.histogram2d`, and `bh.numpy.histogramdd`) are provided. The syntax is identical, though boost-histogram adds three new keyword-only arguments; `storage=` to select the storage, `histogram=bh.Histogram` to produce a boost-histogram instead of a tuple, and `threads=N` to select a number of threads to fill with.

11.2.1 1D histogram example

If you try the following in an IPython session, you will get:

```
import numpy as np
import boost_histogram as bh

norm_vals = np.concatenate(
    [
        np.random.normal(loc=5, scale=1, size=1_000_000),
        np.random.normal(loc=2, scale=0.2, size=200_000),
        np.random.normal(loc=8, scale=0.2, size=200_000),
    ]
)

%%timeit
bins, edges = np.histogram(norm_vals, bins=100, range=(0, 10))
```

```
17.4 ms ± 2.64 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Of course, you then are either left on your own to compute centers, density, widths, and more, or in some cases you can change the computation call itself to add `density=`, or use the matching function inside Matplotlib, and the API is different if you want 2D or ND histograms. But if you already use NumPy histograms and you really don't want to rewrite your code, boost-histogram has adaptors for the three histogram functions in NumPy:

```
%%timeit
bins, edges = bh.numpy.histogram(norm_vals, bins=100, range=(0, 10))
```

```
7.3 ms ± 55.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This is only a hair slower than using the raw boost-histogram API, and is still a nice performance boost over NumPy. You can even use the NumPy syntax if you want a boost-histogram object later:

```
hist = bh.numpy.histogram(norm_vals, bins=100, range=(0, 10), histogram=bh.Histogram)
```

You can later get a NumPy style output tuple from a histogram object:

```
bins, edges = hist.to_numpy()
```

So you can transition your code slowly to boost-histogram.

11.2.2 2D Histogram example

```
data = np.random.multivariate_normal((0, 0), ((1, 0), (0, 0.5)), 10_000_000).T.copy()
```

We can check the performance against NumPy again; NumPy does not do well with regular spaced bins in more than 1D:

```
%%timeit
np.histogram2d(*data, bins=(400, 200), range=((-2, 2), (-1, 1)))
```

```
1.31 s ± 17.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
bh.numpy.histogram2d(*data, bins=(400, 200), range=((-2, 2), (-1, 1)))
```

```
101 ms ± 117 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

For more than one dimension, boost-histogram is more than an order of magnitude faster than NumPy for regular spaced binning. Although optimizations may be added to boost-histogram for common axes combinations later, in 0.6.1, all axes combinations share a common code base, so you can expect *at least* this level of performance regardless of the axes types or number of axes! Threaded filling can give you an even larger performance boost if you have multiple cores and a large fill to perform.

SUBCLASSING (ADVANCED)

Subclassing boost-histogram components is supported, but requires a little extra care to ensure the subclasses do not return un-wrapped boost-histogram components when a subclassed version is available. The issue is that various actions make the C++ -> Python transition over again, such as using `.project()`. For example, let's say you have a `MyHistogram` and a `MyRegular`. If you use `project(0)`, that needs to also return a `MyRegular`, but it is reconverting the return value from C++ to Python, so it has to somehow know that `MyRegular` is the right axis subclass to select from for `MyHistogram`. This is accomplished with families.

When you subclass, you will need to add a family. Any object can be used - the module for your library is a good choice if you only have one “family” of histograms. Boost-histogram uses `boost_histogram`, Hist uses `hist`. You can use anything you want, though; a custom tag object like `MY_FAMILY = object()` works well too. It just has to support `is`, and be the exact same object on all your subclasses.

```
import boost_histogram as bh
import my_package

class Histogram(bh.Histogram, family=my_package):
    ...

class Regular(bh.axis.Regular, family=my_package):
    ...
```

If you only override `Histogram`, you can leave off the `family=` argument, or set it to `None`. It will generate a private `object()` in this case. You must add an explicit family to `Histogram` if you subclass any further components.

If you use Mixins, special care needs to be taken if you need a left-acting Mixin, since class keywords are handled via `super()` left to right. This is a Mixin that will work on either side:

```
class AxisMixin:
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs) # type: ignore
```

Mixins are recommended if you want to provide functionality to a collection of different subclasses, like `Axis`.

There are customization hooks provided for subclasses as well. `self._generate_axes_()` is called to produce an `AxesTuple`, so you can override that if you customize `AxesTuple`.

`_import_bh_` and `_export_bh_` are called when converting an object between histogram libraries. `cls._export_bh_(self)` is called from the outgoing class (being converted from), and `self._import_bh_()` is called afterward on the incoming class (being converted to). So if `h1` is an instance of `H1`, and `H2` is the new class, then `H2(h1)` calls `H1._export_bh_(h2)` and then `h2._import_bh_()` before returning `h2`. The internal repr building for axes is a list produced by `_repr_args_` representing each item in the repr.

COMPARISON WITH BOOST.HISTOGRAM

`boost-histogram` was based on the C++ library `Boost.Histogram`. In most ways, it mimics the spirit of this library; if you learn to use one, you probably can use the other. There are a few differences, however, mostly around adhering to Python conventions:

13.1 Removals

There are a few parts of the `Boost.Histogram` interface that are not bound. They are:

The call operator

This is provided in C++ to allow single item filling, and was designed to mimic the accumulator syntax used elsewhere in Boost. It also works nicely with some STL algorithms. It was not provided in Python because using call to modify an object is not common in Python, using call makes duck-typing more dangerous, and single-item fills are not encouraged in Python due to poor performance. The `.fill` method from `Boost.Histogram 1.72` is bound, however - this provides fast fills without the drawbacks. If you want to fill with a single item, Python's `.fill` does support single item fills.

Histogram make functions

These functions, such as `make_histogram` and `make_weighted_histogram`, are provided in `Boost.Histogram` to make the template syntax easier in C++14. In C++17, they are replaced by directly using the `histogram` constructor; the Python bindings are not limited by old templating syntax, and choose to only provide the newer spelling.

Custom components

Many components in `Boost.Histogram` are configurable or replaceable at compile time; since Python code is precompiled, a comprehensive but static subset was selected for the Python bindings.

13.2 Changes

Naming

The bindings follow modern Python conventions, with CamelCase for classes, etc. The `Boost.Histogram` library follows Boost conventions.

Serialization

The Python bindings use a pickle-based binary serialization, so cannot read files saved in C++ using `Boost.Serialize`.

Properties

Many methods in C++ are properties in Python. `.axis(i)` is replaced with `.axes[i]`.

Indexing

The Python bindings use standard Python indexing for selection and setting. You can recover the functionality of `.at(i)` at endpoints with `bh.tag.at(i)`.

Renames

The `.rank()` method is replaced by the `.ndim` property to match the common NumPy spelling.

13.3 Additions

Unified Histogram Indexing

The Python bindings support UHI, a proposal to unify and simplify histogram indexing in Python.

Custom transforms

Custom transforms are possible using Numba or a C pointer. In Boost.Histogram, you can use templating to make arbitrary transforms, so a run time transform is not as necessary (but may be added).

NumPy compatibility

The Python bindings do several things to simplify NumPy compatibility.

SIMPLE EXAMPLE

Let's try some basic functionality of boost-histogram:

```
[1]: import numpy as np

import boost_histogram as bh

[2]: vals = np.random.normal(size=(2, 1_000_000))

[3]: hist = bh.Histogram(
    bh.axis.Regular(10, 0, 10, metadata="x", transform=bh.axis.transform.sqrt),
    bh.axis.Regular(10, 0, 1, circular=True, metadata="y"),
    storage=bh.storage.Int64(),
)
hist

[3]: Histogram(
    Regular(10, 0, 10, metadata='x', transform=sqrt),
    Regular(10, 0, 1, circular=True, metadata='y'),
    storage=Int64())
```

This fills the histogram

```
[4]: hist.fill(*vals)

[4]: Histogram(
    Regular(10, 0, 10, metadata='x', transform=sqrt),
    Regular(10, 0, 1, circular=True, metadata='y'),
    storage=Int64()) # Sum: 499682.0 (1000000.0 with flow)
```

Let's just take a quick look at the bin contents:

```
[5]: print(hist.view())

[[ 3955  3992  3972  3948  3930  3987  4073  3982  3878  4073]
 [11519 11675 11560 11528 11612 11575 11476 11643 11573 11603]
 [16197 16087 16261 16043 15775 15912 16067 16010 15973 15998]
 [12843 12956 12855 12912 12870 12940 12962 12891 12752 13073]
 [ 4807  4952  4844  4859  4869  4788  4657  4931  4775  4874]
 [   636    618    598    608    633    644    603    635    611    645]
 [    14     18     20     13     12     21     10     18     15     22]
 [     0      0      0      0      1      0      0      0      0      0]]
```

(continues on next page)

(continued from previous page)

[0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0]]

ROOT FILE FORMAT EXAMPLE

To run this example, you will need `uproot`, which is another SciKit-HEP library.

```
[1]: from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import uproot

import boost_histogram as bh

demo_file = Path("demo_uproot_file.root")
```

`ROOT` is a modular scientific software toolkit used in High Energy Physics. (HEP) The `ROOT` file format is used to store almost all HEP data. This notebook will illustrate one method for converting to/from the `ROOT` file format using `uproot`, a Python implementation of a `ROOT` file reader and writer.

For more complicated histograms, you may need `Aghast` and `PyROOT`, but that is a much heavier dependency, and is covered in a separate tutorial.

Start by making a 1D histogram:

```
[2]: h = bh.Histogram(bh.axis.Regular(15, -3, 3))
h.fill(np.random.normal(size=1_000_000))

[2]: Histogram(Regular(15, -3, 3), storage=Double()) # Sum: 997352.0 (1000000.0 with flow)

[3]: with uproot.recreate(demo_file) as root_file:
    # Uproot automatically converts histograms
    root_file["hist"] = h
```

If you want to save and load the `ROOT` histogram, use `uproot` to read and write:

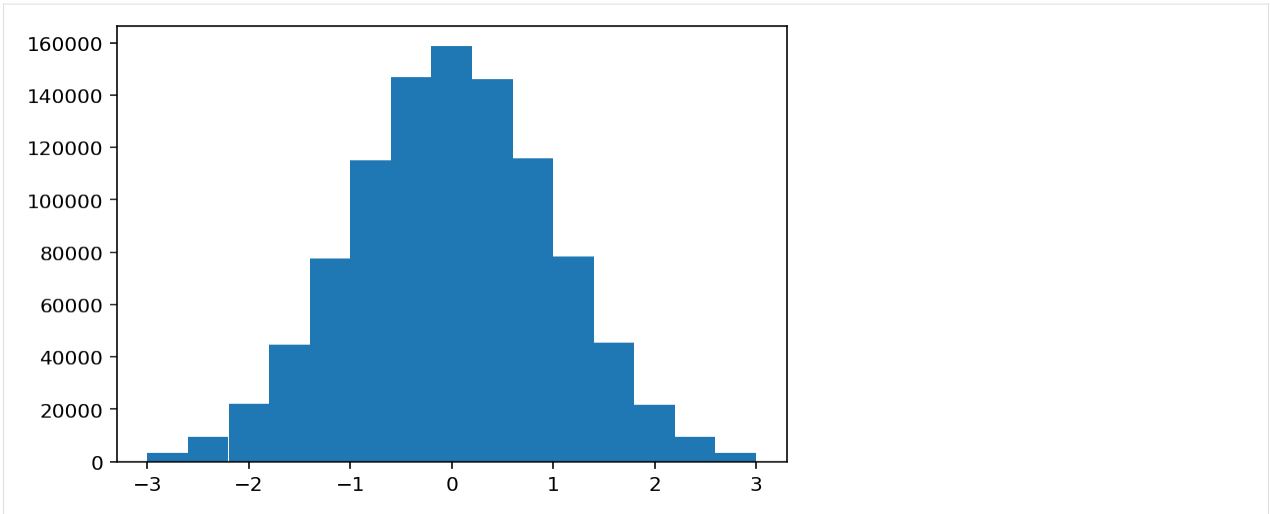
```
[4]: with uproot.open(demo_file) as root_file_2:
    uproot_hist = root_file_2["hist"]

print(uproot_hist)

<TH1D (version 3) at 0x7fa9b83ffd00>
```

This `uproot` histogram can be converted directly to `boost_histogram`:

```
[5]: h = bh.Histogram(uproot_hist)
plt.bar(h.axes[0].centers, h.values(), width=h.axes[0].widths);
```



We could use a `Weight()` storage and read both `allvalues` and `allvariances` in, as well, since ROOT histograms can sometimes have this enabled.

Finally, let's clean up after ourselves:

```
[6]: if demo_file.is_file():  
      demo_file.unlink()
```


THREADED FILLS

```
[1]: from concurrent.futures import ThreadPoolExecutor
    from functools import reduce
    from operator import add

    import numpy as np
    from numpy.testing import assert_array_equal

    import boost_histogram as bh
```

This notebook explores parallel filling by hand (not using the `threads=` argument).

```
[2]: hist_linear = bh.Histogram(bh.axis.Regular(100, 0, 1))
    hist_atomic = bh.Histogram(bh.axis.Regular(100, 0, 1), storage=bh.storage.AtomicInt64())

    vals = np.random.rand(10_000_000)
    hist_answer = hist_linear.fill(vals).copy()
```

This is a traditional fill.

```
[3]: %%timeit
    hist_linear.reset()
    hist_linear.fill(vals)
    assert_array_equal(hist_answer, hist_linear)

25.5 ms ± 774 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

This is a single threaded atomic fill.

```
[4]: %%timeit
    hist_atomic.reset()
    hist_atomic.fill(vals)
    assert_array_equal(hist_answer, hist_atomic)

59.9 ms ± 2.19 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

This is a threaded fill (storage not threadsafe, so will get the wrong result; just for comparison)

This is a threaded fill, this time with atomics. It may not be faster, but is useful in situations where you are filling from multiple places in your code.

```
[5]: %%timeit
    hist_atomic.reset()
```

(continues on next page)

(continued from previous page)

```
threads = 4
with ThreadPoolExecutor(threads) as pool:
    for chunk in np.array_split(vals, threads):
        pool.submit(hist_atomic.fill, chunk)
assert_array_equal(hist_answer, hist_atomic)
```

61.2 ms \pm 1.57 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)

This makes four separate histograms, then fills them and adds at the end.

```
[6]: def fun(x):
      hist = bh.Histogram(bh.axis.Regular(100, 0, 1))
      return hist.fill(x)
```

```
[7]: %%timeit
      threads = 4
      with ThreadPoolExecutor(threads) as pool:
          results = pool.map(fun, np.array_split(vals, threads))
      hist_quad = reduce(add, results)
      assert_array_equal(hist_answer, hist_quad)
```

8.12 ms \pm 90 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

The expense of creating the histogram and summing them must be significantly less than the cost of filling for this to be faster.

PERFORMANCE COMPARISON

We will compare boost-histogram to numpy.

```
[1]: import numpy as np
    from numpy.testing import assert_allclose

    import boost_histogram as bh
```

```
[2]: import os

    threads = os.cpu_count() // 2
    print(f"threads: {threads}")

threads: 8
```

17.1 Testing setup

This is just a simple 1D and 2D dataset to use for performance runs. The testing setup is the same as “MBP” in [this post](#), a dual-core MacBook Pro 2015.

```
[3]: bins = (100, 100)
    ranges = ((-3, 3), (-3, 3))
    bins = np.asarray(bins).astype(np.int64)
    ranges = np.asarray(ranges).astype(np.float64)

    edges = (
        np.linspace(*ranges[0, :], bins[0] + 1),
        np.linspace(*ranges[1, :], bins[1] + 1),
    )
```

```
[4]: np.random.seed(42)
    vals = np.random.normal(size=[2, 10_000_000]).astype(np.float32)
    vals1d = np.random.normal(size=[10_000_000]).astype(np.float32)
```

17.1.1 Traditional 1D NumPy Histogram

This is reasonably optimized; it should provide good performance.

```
[5]: answer, e = np.histogram(vals1d, bins=bins[0], range=ranges[0])
```

```
[6]: %%timeit
h, _ = np.histogram(vals1d, bins=bins[0], range=ranges[0])
assert_allclose(h, answer, atol=1)

74.5 ms ± 2.37 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

17.1.2 Boost histogram 1D

```
[7]: %%timeit
hist = bh.Histogram(bh.axis.Regular(bins[0], *ranges[0]), storage=bh.storage.Int64())
hist.fill(vals1d)
assert_allclose(hist, answer, atol=1)

41.6 ms ± 712 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

17.1.3 Boost histogram 1D NumPy clone

```
[8]: %%timeit
h, _ = bh.numpy.histogram(vals1d, bins=bins[0], range=ranges[0])
assert_allclose(h, answer, atol=1)

43.1 ms ± 769 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

17.1.4 Boost histogram in 1D, threaded

```
[9]: %%timeit
hist = bh.Histogram(bh.axis.Regular(bins[0], *ranges[0]), storage=bh.storage.Int64())

hist.fill(vals1d, threads=threads)
assert_allclose(hist, answer, atol=1)

13.3 ms ± 153 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

17.1.5 Boost histogram 1D NumPy clone, threaded

```
[10]: %%timeit
h, _ = bh.numpy.histogram(vals1d, bins=bins[0], range=ranges[0], threads=threads)
assert_allclose(h, answer, atol=1)

13.8 ms ± 238 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

17.1.6 Traditional 2D NumPy histogram

Not as well optimized for regular filling.

```
[11]: answer2, *ledges = np.histogram2d(*vals, bins=bins, range=ranges)
```

```
[12]: %%timeit
H, *ledges = np.histogram2d(*vals, bins=bins, range=ranges)
assert_allclose(H, answer2, atol=1)

874 ms ± 22.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

17.1.7 Boost histogram in 2D

```
[13]: %%timeit
hist = bh.Histogram(
    bh.axis.Regular(bins[0], *ranges[0]), bh.axis.Regular(bins[1], *ranges[1])
)
hist.fill(*vals)
assert_allclose(hist, answer2, atol=1)

77.6 ms ± 615 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

17.1.8 Boost histogram 2D NumPy clone

```
[14]: %%timeit
H, *ledges = bh.numpy.histogram2d(*vals, bins=bins, range=ranges)
assert_allclose(H, answer2, atol=1)

84.7 ms ± 2.78 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

17.1.9 Boost histogram in 2D, threaded

```
[15]: %%timeit
hist = bh.Histogram(
    bh.axis.Regular(bins[0], *ranges[0]), bh.axis.Regular(bins[1], *ranges[1])
)

hist.fill(*vals, threads=threads)
assert_allclose(hist, answer2, atol=1)

28.7 ms ± 708 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

17.1.10 Boost histogram 2D NumPy clone, threaded

```
[16]: %%timeit
H, *ledges = bh.numpy.histogram2d(*vals, bins=bins, range=ranges, threads=threads)
assert_allclose(H, answer2, atol=1)
```

29.6 ms \pm 503 μ s per loop (mean \pm std. dev. of 7 runs, 10 loops each)

XARRAY EXAMPLE

To run this example, an environment.yml similar to this one could be used:

```
name: bh_xhistogram
channels:
  - conda-forge
dependencies:
  - python==3.8
  - boost-histogram
  - xhistogram
  - matplotlib
  - netcdf4
```

```
[1]: import matplotlib.pyplot as plt
import numpy as np
import xarray as xr
from xhistogram.xarray import histogram as xhistogram

import boost_histogram as bh
```

Let's look at using boost-histogram to imitate the xhistogram package by reading and producing xarrays. As a reminder, xarray is a sort of generalized Pandas library, supporting ND labeled and indexed data.

18.1 Simple 1D example

We will start with the first example from the xhistogram docs:

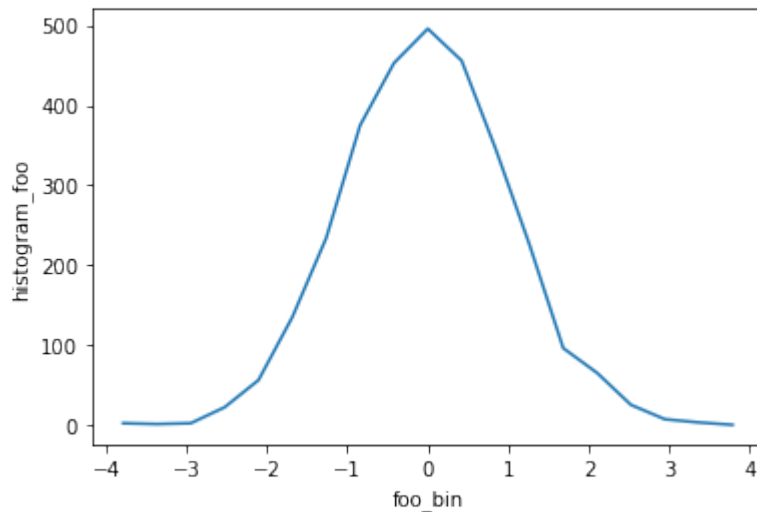
```
[2]: da = xr.DataArray(np.random.randn(100, 30), dims=["time", "x"], name="foo")
bins = np.linspace(-4, 4, 20)
```

18.1.1 xhistogram

And, this is what historamming and plotting looks like:

```
[3]: h = xhistogram(da, bins=[bins])
      display(h)
      h.plot()
      # h is an xarray
```

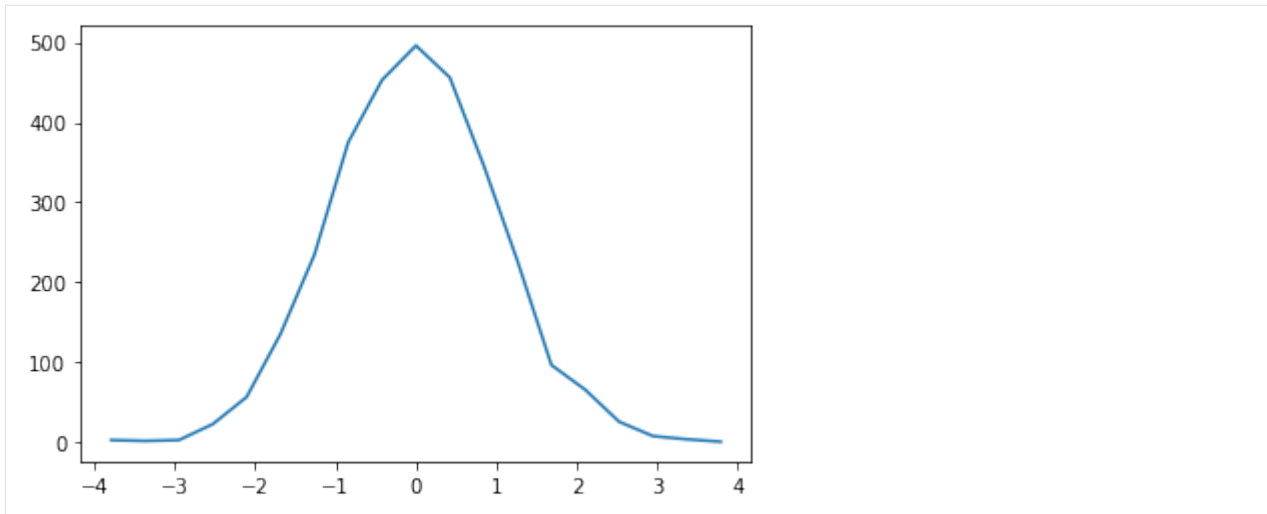
```
<xarray.DataArray 'histogram_foo' (foo_bin: 19)>
array([  2,   1,   2,  22,  56, 135, 234, 375, 453, 496, 456, 346, 226,
        96,  65,  25,   7,   3,   0])
Coordinates:
  * foo_bin  (foo_bin) float64 -3.789 -3.368 -2.947 -2.526 ... 2.947 3.368 3.789
```



18.1.2 boost-histogram (direct usage)

Let's first just try this by hand, to see how it works. This will not return an xarray, etc.

```
[4]: bh_bins = bh.axis.Regular(19, -4, 4)
      bh_hist = bh.Histogram(bh_bins).fill(np.asarray(da).flatten())
      plt.plot(bh_hist.axes[0].centers, bh_hist.values());
```

18.1.3 boost-histogram (adapter function)

Now, let's make an adaptor for boost-histogram.

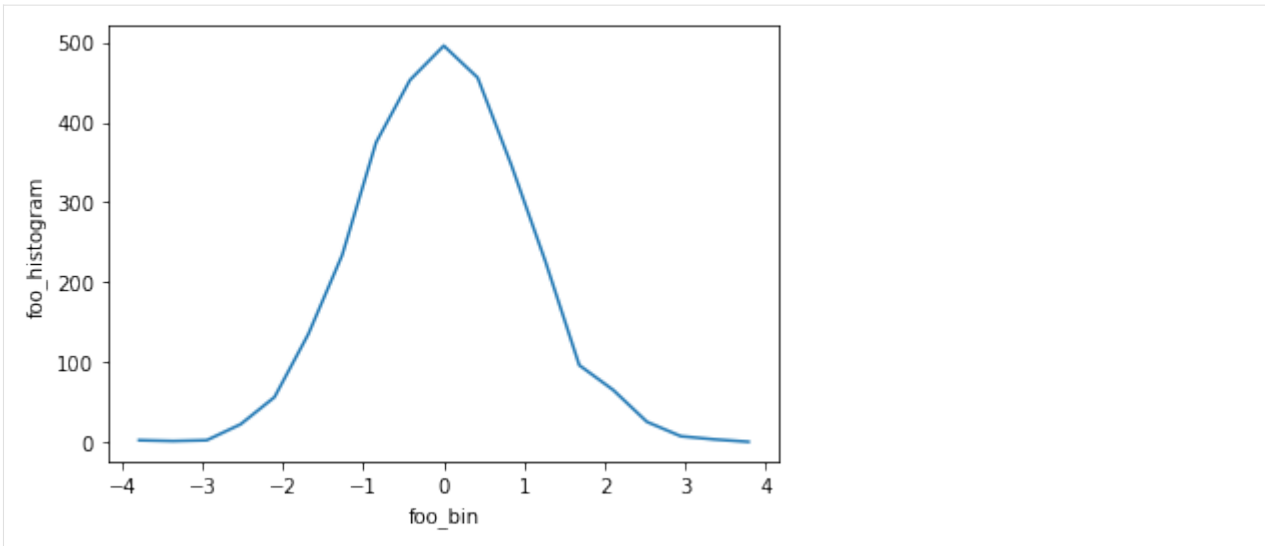
```
[5]: def bh_xhistogram(*args, bins):
    # Convert bins to boost-histogram axes first
    prepare_bins = (bh.axis.Variable(b) for b in bins)
    h = bh.Histogram(*prepare_bins)

    # We need flat NP arrays for filling
    prepare_fill = (np.asarray(a).flatten() for a in args)
    h.fill(*prepare_fill)

    # Now compute the xarray output.
    return xr.DataArray(
        h.values(),
        name=".".join(a.name for a in args) + "_histogram",
        coords=[
            (f"{a.name}_bin", arr.flatten(), a.attrs)
            for a, arr in zip(args, h.axes.centers)
        ],
    )
```

```
[6]: h = bh_xhistogram(da, bins=[bins])
display(h)
h.plot();

<xarray.DataArray 'foo_histogram' (foo_bin: 19)>
array([ 2.,  1.,  2., 22., 56., 135., 234., 375., 453., 496., 456.,
        346., 226., 96., 65., 25.,  7.,  3.,  0.])
Coordinates:
  * foo_bin  (foo_bin) float64 -3.789 -3.368 -2.947 -2.526 ... 2.947 3.368 3.789
```



More features

Let's add a few more features to our function defined above. * Let's allow bins to be a list of axes or even a completely prepared histogram; this will allow us to take advantage of boost-histogram features later. * Let's add a weights keyword so we can do weighted histograms as well.

```
[7]: def bh_xhistogram(*args, bins, weights=None):
    """
    bins is either a histogram, a list of axes, or a list of bins
    """

    if isinstance(bins, bh.Histogram):
        h = bins
    else:
        prepare_bins = (
            b if isinstance(b, bh.axis.Axis) else bh.axis.Variable(b) for b in bins
        )
        h = bh.Histogram(*prepare_bins)

    prepare_fill = (np.asarray(a).flatten() for a in args)

    if weights is None:
        h.fill(*prepare_fill)
    else:
        prepared_weights, *_ = xr.broadcast(weights, *args)
        h.fill(*prepare_fill, weight=np.asarray(prepared_weights).flatten())

    return xr.DataArray(
        h.values(),
        name="_".join(a.name for a in args) + "_histogram",
        coords=[
            (f"{a.name}_bin", arr.flatten(), a.attrs)
            for a, arr in zip(args, h.axes.centers)
        ],
```

(continues on next page)

(continued from previous page)

)

18.2 2D example

This also comes from the xhistogram docs.

```
[8]: # Read WOA using opendap
Temp_url = (
    "http://apdrc.soest.hawaii.edu:80/dods/public_data/WOA/WOA13/5_deg/annual/temp"
)
Salt_url = (
    "http://apdrc.soest.hawaii.edu:80/dods/public_data/WOA/WOA13/5_deg/annual/salt"
)
Oxy_url = (
    "http://apdrc.soest.hawaii.edu:80/dods/public_data/WOA/WOA13/5_deg/annual/doxy"
)

ds = xr.merge(
    [
        xr.open_dataset(Temp_url).tmn.load(),
        xr.open_dataset(Salt_url).smn.load(),
        xr.open_dataset(Oxy_url).omn.load(),
    ]
)
```

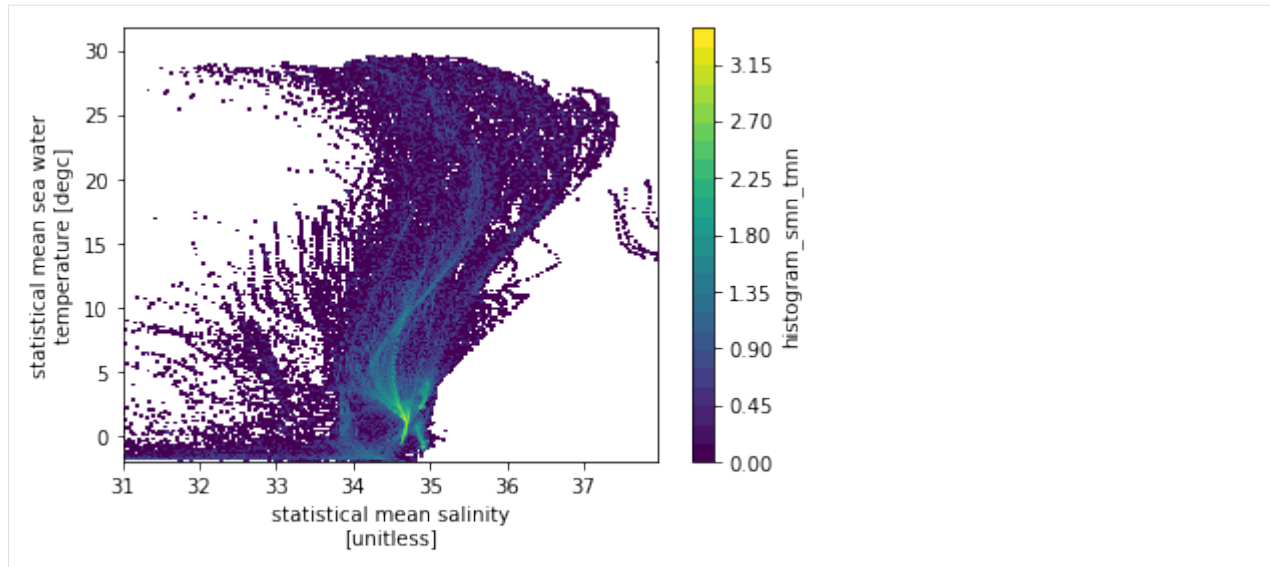
```
[9]: sbins = np.arange(31, 38, 0.025)
     tbins = np.arange(-2, 32, 0.1)
```

18.2.1 xhistogram

```
[10]: hTS = xhistogram(ds.smn, ds.tmn, bins=[sbins, tbins])
      np.log10(hTS.T).plot(levels=31)

/usr/local/Caskroom/miniconda/base/envs/xtest/lib/python3.8/site-packages/xarray/core/
↳ computation.py:601: RuntimeWarning: divide by zero encountered in log10
    result_data = func(*input_data)

[10]: <matplotlib.collections.QuadMesh at 0x7f84c920b130>
```



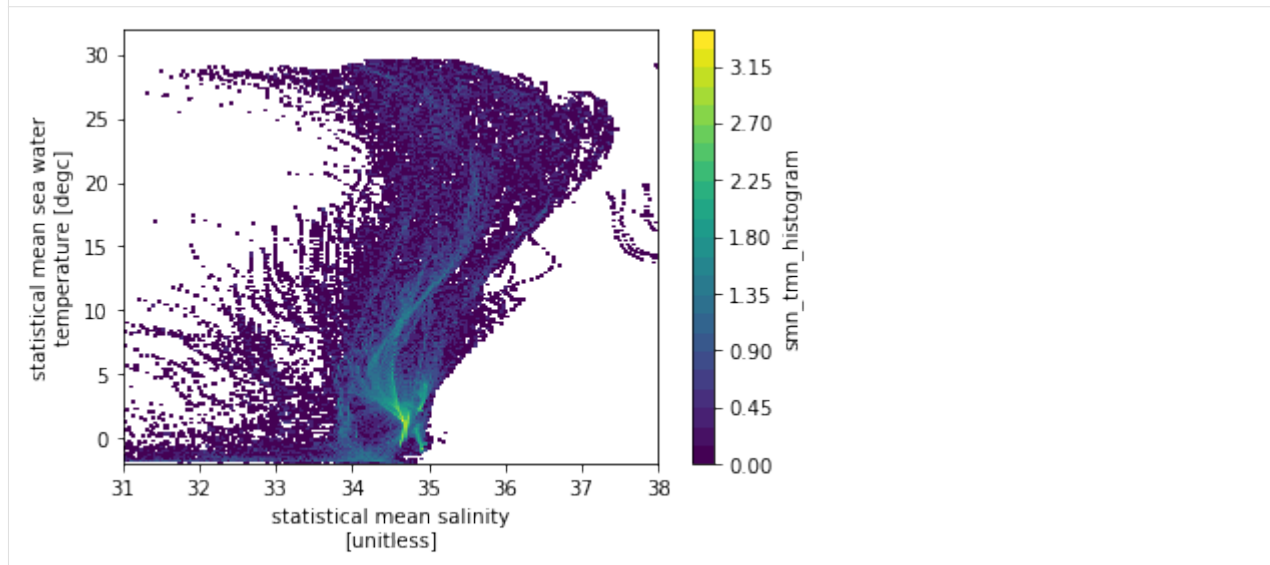
18.2.2 boost-histogram

We could hand in the same bin definitions, but let's use boost-histogram axes instead:

```
[11]: sax = bh.axis.Regular(250, 31, 38)
      tax = bh.axis.Regular(340, -2, 32)

      hTS = bh_xhistogram(ds.smn, ds.tmn, bins=[sax, tax])
      np.log10(hTS.T).plot(levels=31)
```

```
[11]: <matplotlib.collections.QuadMesh at 0x7f8528c132b0>
```



Speed comparson

```
[12]: %%timeit
hTS = xhistogram(ds.smn, ds.tmn, bins=[sbins, tbins])

17.6 ms ± 508 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
[13]: %%timeit
hTS = bh_xhistogram(ds.smn, ds.tmn, bins=[sax, tax])

4.7 ms ± 245 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Weighted histogram

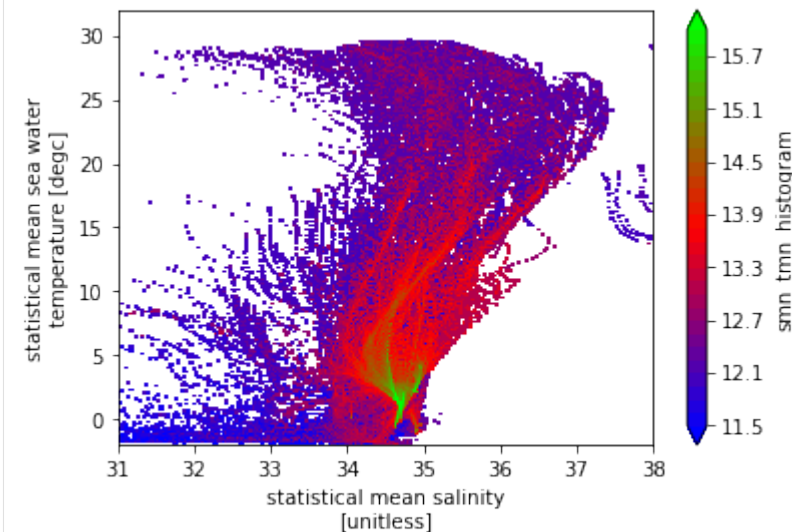
Let's try a more complex example from the docs; the dVol weights one:

```
[14]: dz = np.diff(ds.lev)
dz = np.insert(dz, 0, dz[0])
dz = xr.DataArray(dz, coords={"lev": ds.lev}, dims="lev")

dVol = dz * (5 * 110e3) * (5 * 110e3 * np.cos(ds.lat * np.pi / 180))

hTSw = bh_xhistogram(ds.smn, ds.tmn, bins=[sax, tax], weights=dVol)
np.log10(hTSw.T).plot(levels=31, vmin=11.5, vmax=16, cmap="brg")
```

```
[14]: <matplotlib.collections.QuadMesh at 0x7f84b821c940>
```



USING BOOST-HISTOGRAM

```
[1]: import functools
import operator

import matplotlib.pyplot as plt
import numpy as np

import boost_histogram as bh
```

19.1 1: Basic 1D histogram

Let's start with the basics. We will create a histogram using boost-histogram and fill it.

19.1.1 1.1: Data

Let's make a 1d dataset to run on.

```
[2]: data1 = np.random.normal(3.5, 2.5, size=1_000_000)
```

Now, let's make a histogram

```
[3]: hist1 = bh.Histogram(bh.axis.Regular(40, -2, 10))
```

```
[4]: hist1.fill(data1)
```

```
[4]: Histogram(Regular(40, -2, 10), storage=Double()) # Sum: 981542.0 (1000000.0 with flow)
```

You can see that the histogram has been filled. Let's explicitly check to see how many entries are in the histogram:

```
[5]: hist1.sum()
```

```
[5]: 981542.0
```

What happened to the missing items? They are in the underflow and overflow bins:

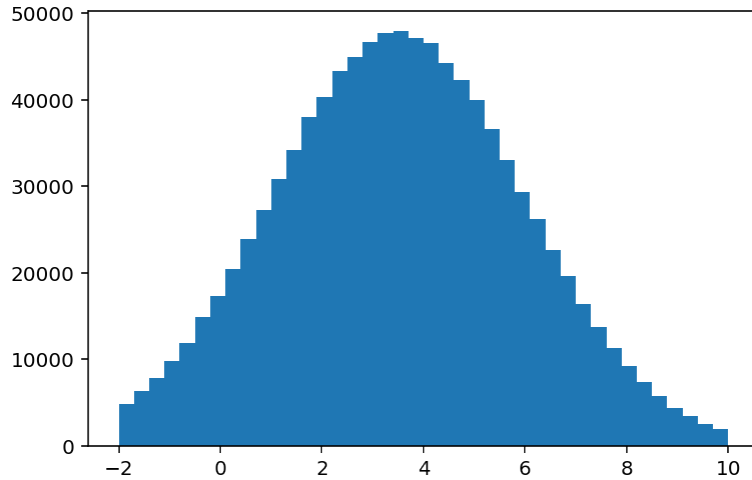
```
[6]: hist1.sum(flow=True)
```

```
[6]: 1000000.0
```

Like ROOT, we have overflow bins by default. We can turn them off, but they enable some powerful things like projections.

Let's plot this (Hist should make this easier):

```
[7]: plt.bar(hist1.axes[0].centers, hist1.values(), width=hist1.axes[0].widths);
```



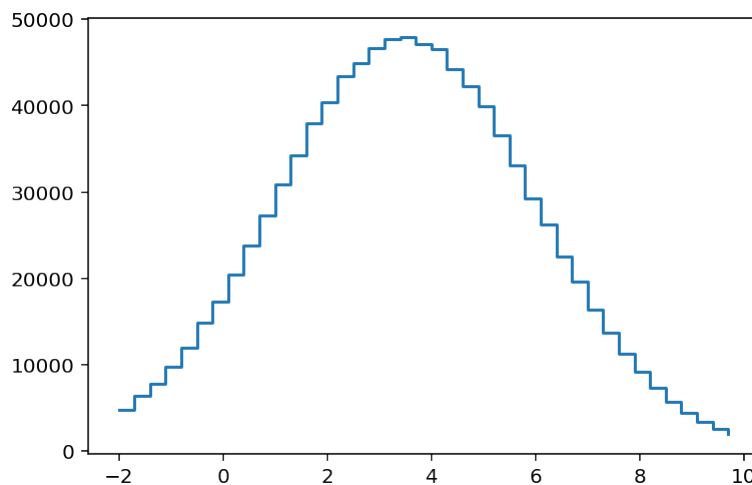
Note: you can select the axes before or after calling `.centers`; this is very useful for ND histograms.

From now on, let's be lazy

```
[8]: plothist = lambda h: plt.bar(*h.axes.centers, h.values(), width=h.axes.widths[0]);
```

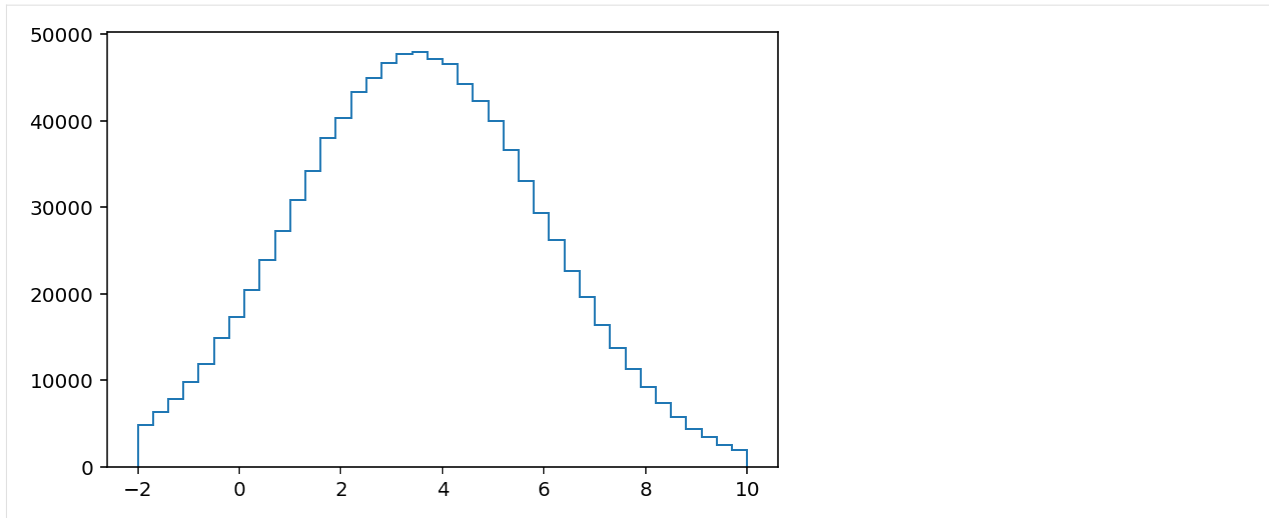
Aside: here's step. The edges are quite ugly for us, just like it is for numpy. Or anyone.

```
[9]: plt.step(hist1.axes[0].edges[:-1], hist1.values(), where="post");
```



Recent versions of matplotlib support `.stairs`, which was designed to work well with histograms:

```
[10]: plt.stairs(hist1.values(), hist1.axes[0].edges);
```

No plotting is built in, but the data is easy to access.

19.2 2: Drop-in replacement for NumPy

To start using this yourself, you don't even need to change your code. Let's try the numpy adapters.

```
[11]: bins2, edges2 = bh.numpy.histogram(data1, bins=10)
```

```
[12]: b2, e2 = np.histogram(data1, bins=10)
```

```
[13]: bins2 - b2
```

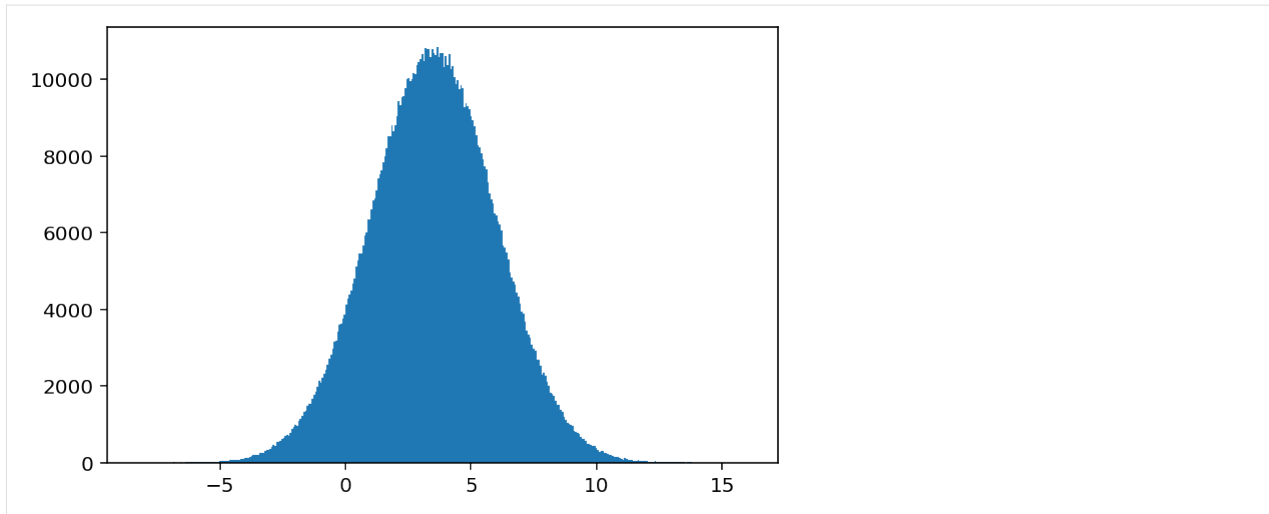
```
[13]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int64)
```

```
[14]: e2 - edges2
```

```
[14]: array([ 0.00000000e+00,  8.88178420e-16,  8.88178420e-16,  0.00000000e+00,
            0.00000000e+00,  0.00000000e+00,  1.77635684e-15,  3.55271368e-15,
            0.00000000e+00,  0.00000000e+00, -1.77635684e-15])
```

Not bad! Let's start moving to the boost-histogram API, so we can use our little plotting function:

```
[15]: hist2 = bh.numpy.histogram(data1, bins="auto", histogram=bh.Histogram)
      plothist(hist2);
```



Now we can move over to boost-histogram one step at a time! Just to be complete, we can also go back to a NumPy tuple from a Histogram object:

```
[16]: b2p, e2p = bh.numpy.histogram(data1, bins=10, histogram=bh.Histogram).to_numpy()
      b2p == b2
[16]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True])
```

19.3 3: More dimensions

The same API works for multiple dimensions.

```
[17]: hist3 = bh.Histogram(bh.axis.Regular(150, -1.5, 1.5), bh.axis.Regular(100, -1, 1))

[18]: def make_2D_data(*, mean=(0, 0), widths=(1, 1), size=1_000_000):
      cov = np.asarray(widths) * np.eye(2)
      return np.random.multivariate_normal(mean, cov, size=size).T

[19]: data3x = make_2D_data(mean=[-0.75, 0.5], widths=[0.2, 0.02])
      data3y = make_2D_data(mean=[0.75, 0.5], widths=[0.2, 0.02])
```

From here on out, I will be using `.reset()` before a `.fill()`, just to make sure each cell in the notebook can be rerun.

```
[20]: hist3.reset()
      hist3.fill(*data3x)
      hist3.fill(*data3y)

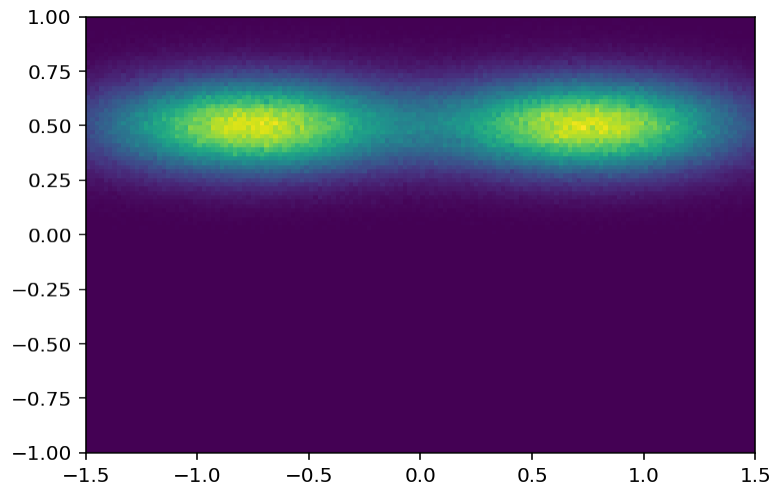
[20]: Histogram(
  Regular(150, -1.5, 1.5),
  Regular(100, -1, 1),
  storage=Double()) # Sum: 1905785.0 (20000000.0 with flow)
```

Again, let's make plotting a little function:

```
[21]: def plothist2d(h):
        return plt.pcolormesh(*h.axes.edges.T, h.values().T)
```

This is transposed because `pcolormesh` expects it.

```
[22]: plothist2d(hist3);
```



Let's try a 3D histogram

```
[23]: data3d = [np.random.normal(size=1_000_000) for _ in range(3)]

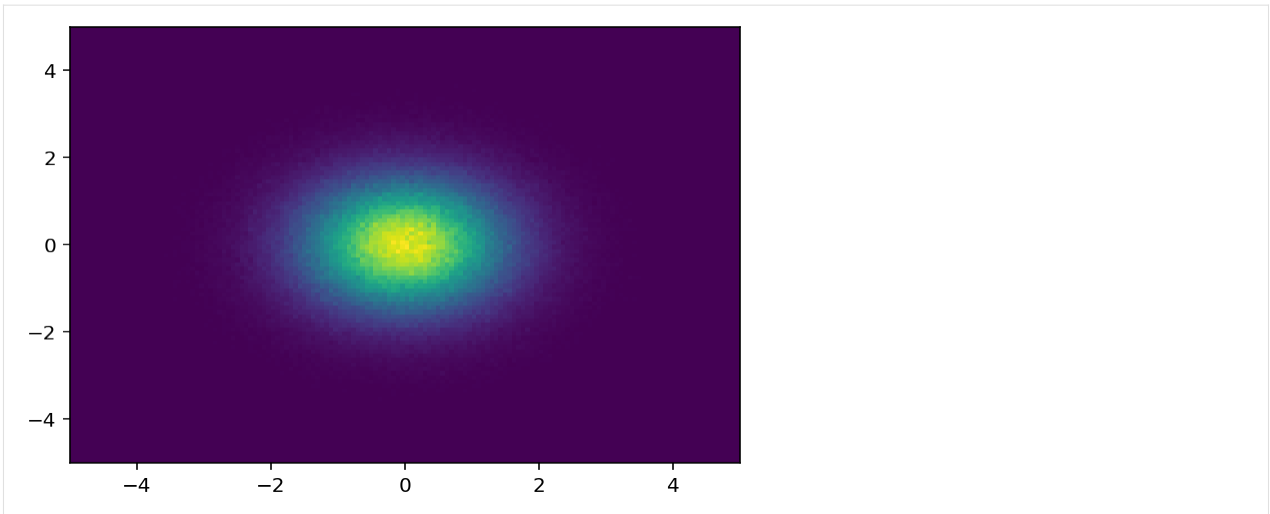
hist3d = bh.Histogram(
    bh.axis.Regular(150, -5, 5),
    bh.axis.Regular(100, -5, 5),
    bh.axis.Regular(100, -5, 5),
)

hist3d.fill(*data3d)
```

```
[23]: Histogram(
    Regular(150, -5, 5),
    Regular(100, -5, 5),
    Regular(100, -5, 5),
    storage=Double()) # Sum: 1000000.0
```

Let's project to the first two axes:

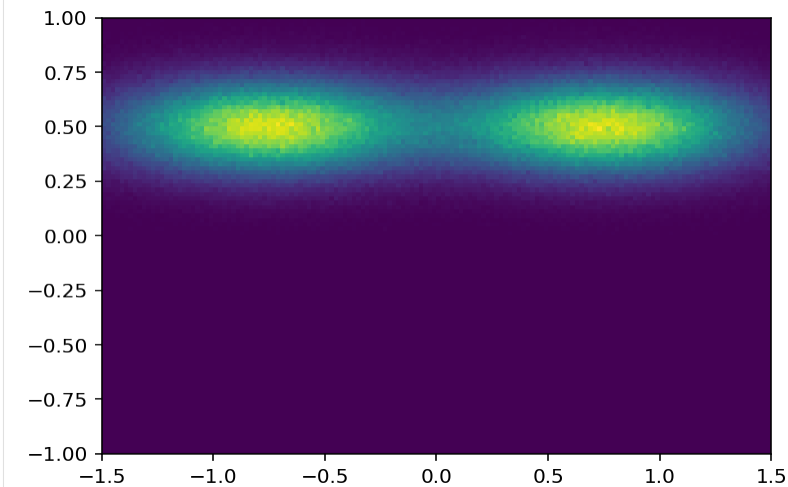
```
[24]: plothist2d(hist3d.project(0, 1));
```



19.4 4: UHI

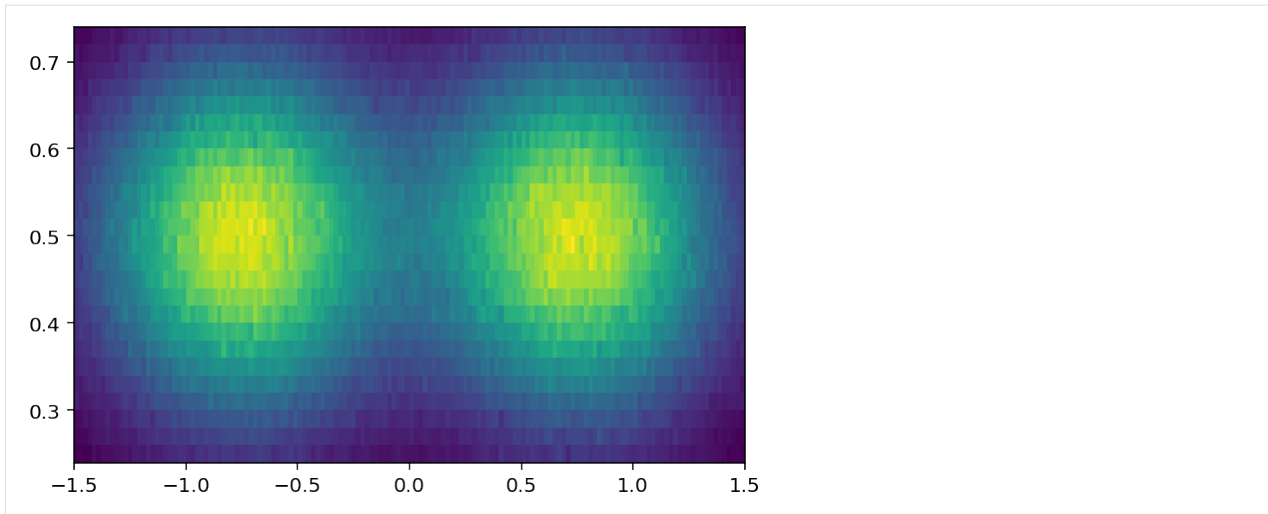
Let's explore the boost-histogram UHI syntax. We will reuse the previous 2D histogram from part 3:

```
[25]: plothist2d(hist3);
```



I can see that I want y from 0.25 to 0.75, in data coordinates:

```
[26]: plothist2d(hist3[:, bh.loc(0.25) : bh.loc(0.75)]);
```



What's the contents of a bin?

```
[27]: hist3[100, 87]
```

```
[27]: 181.0
```

How about in data coordinates?

```
[28]: hist3[bh.loc(0.5), bh.loc(0.75)]
```

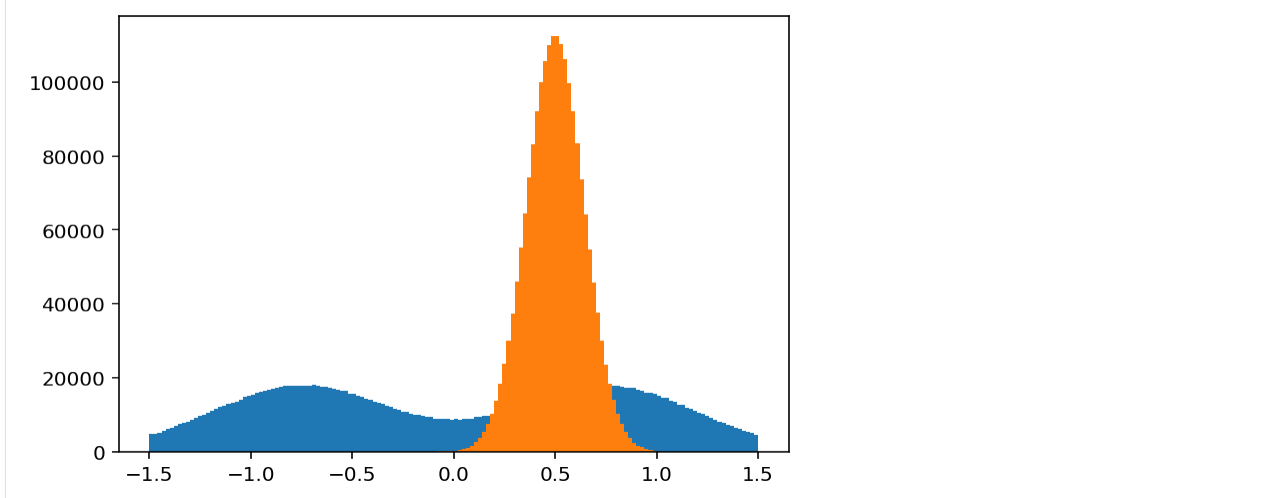
```
[28]: 181.0
```

Note: to get the coordinates manually:

```
hist3.axes[0].index(.5) == 100
hist3.axes[1].index(.75) == 87
```

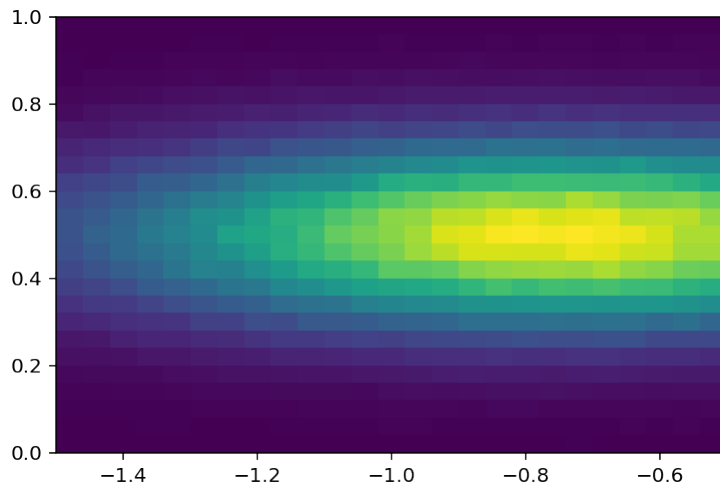
How about a 1d histogram?

```
[29]: plothist(hist3[:, :: bh.sum])
      plothist(hist3[:, : bh.sum, :]);
```



Let's look at one part and rebin:

```
[30]: plothist2d(hist3[: 50 : bh.rebin(2), 50 :: bh.rebin(2)]);
```



What is the value at $(-0.75, 0.5)$?

```
[31]: hist3[bh.loc(-0.75), bh.loc(0.5)]
```

```
[31]: 1005.0
```

19.5 5: Understanding accumulators

Boost-histogram has several different storages; storages store accumulators. Let's try making a profile.

```
[32]: mean = bh.accumulators.Mean()
      mean.fill([0.3, 0.4, 0.5])
```

```
[32]: Mean(count=3, value=0.4, variance=0.01)
```

Here's a quick example accessing the values:

```
[33]: print(
      f"mean.count={mean.count} mean.value={mean.value:g} mean.variance={mean.variance:g}"
    )
```

```
# Python 3.8:
# print(f"{mean.count=} {mean.value=} {mean.variance=}")
```

```
mean.count=3.0 mean.value=0.4 mean.variance=0.01
```

19.6 6: Changing the storage

```
[34]: hist6 = bh.Histogram(bh.axis.Regular(10, 0, 10), storage=bh.storage.Mean())

[35]: hist6.fill([0.5] * 3, sample=[0.3, 0.4, 0.5])

[35]: Histogram(Regular(10, 0, 10), storage=Mean()) # Sum: Mean(count=3, value=0.4, variance=0.
↪ 01)

[36]: hist6[0]

[36]: Mean(count=3, value=0.4, variance=0.01)

[37]: hist6.view()

[37]: MeanView(
    [(3., 0.4, 0.02), (0., 0. , 0. ), (0., 0. , 0. ), (0., 0. , 0. ),
     (0., 0. , 0. ), (0., 0. , 0. ), (0., 0. , 0. ), (0., 0. , 0. ),
     (0., 0. , 0. ), (0., 0. , 0. )],
    dtype=[('count', '<f8'), ('value', '<f8'), ('_sum_of_deltas_squared', '<f8')])

[38]: hist6.view().value

[38]: array([0.4, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ])

[39]: hist6.view().variance

[39]: array([ 0.01, -0. , -0. , -0. , -0. , -0. , -0. , -0. , -0. ,
            -0.  ])
```

19.7 7: Making a density histogram

Let's try to make a density histogram like NumPy's.

```
[40]: bins = [
    -10,
    -7,
    -4,
    -3,
    -2,
    -1,
    -0.75,
    -0.5,
    -0.25,
    0,
    0.25,
    0.5,
    0.75,
    1,
    2,
    3,
```

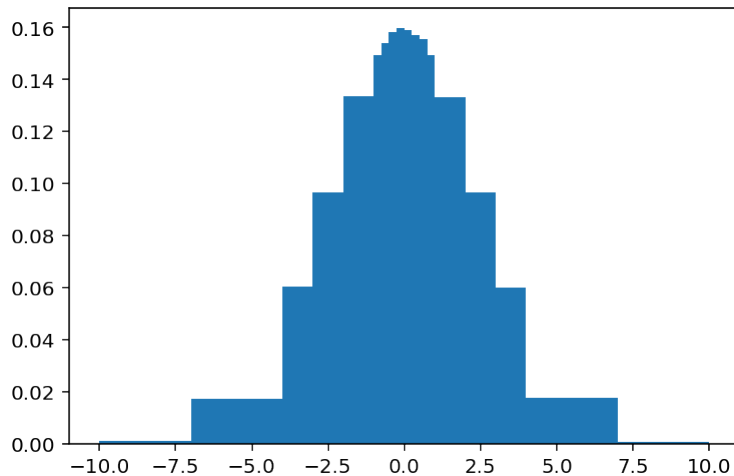
(continues on next page)

(continued from previous page)

```

4,
7,
10,
]
d7, e7 = np.histogram(data1 - 3.5, bins=bins, density=True)
plt.hist(data1 - 3.5, bins=bins, density=True);

```



Yes, it's ugly. Don't judge.

We don't have a `.density`! What do we do? (note: `density=True` is supported if you do not return a `bh` object)

```
[41]: hist7 = bh.numpy.histogram(data1 - 3.5, bins=bins, histogram=bh.Histogram)
```

```

widths = hist7.axes.widths
area = functools.reduce(operator.mul, hist7.axes.widths)

area

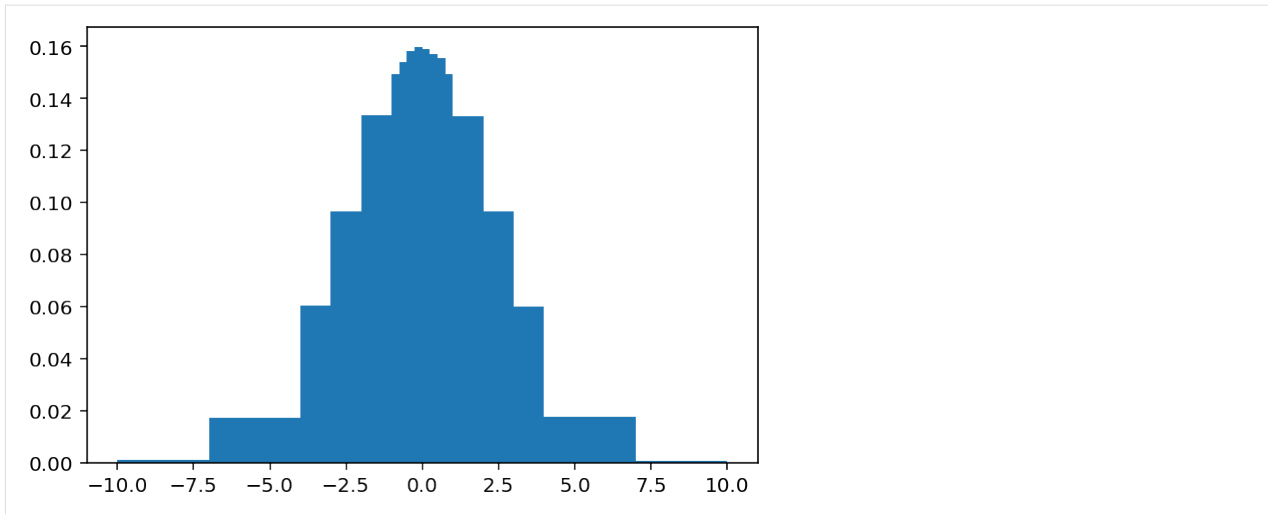
```

```
[41]: array([3. , 3. , 1. , 1. , 1. , 0.25, 0.25, 0.25, 0.25, 0.25, 0.25,
          0.25, 0.25, 1. , 1. , 1. , 3. , 3. ])
```

Yes, that does not need to be so complicated for 1D, but it's general.

```
[42]: factor = np.sum(hist7.values())
view = hist7.values() / (factor * area)
```

```
[43]: plt.bar(hist7.axes[0].centers, view, width=hist7.axes[0].widths);
```

19.8 8: Axis types

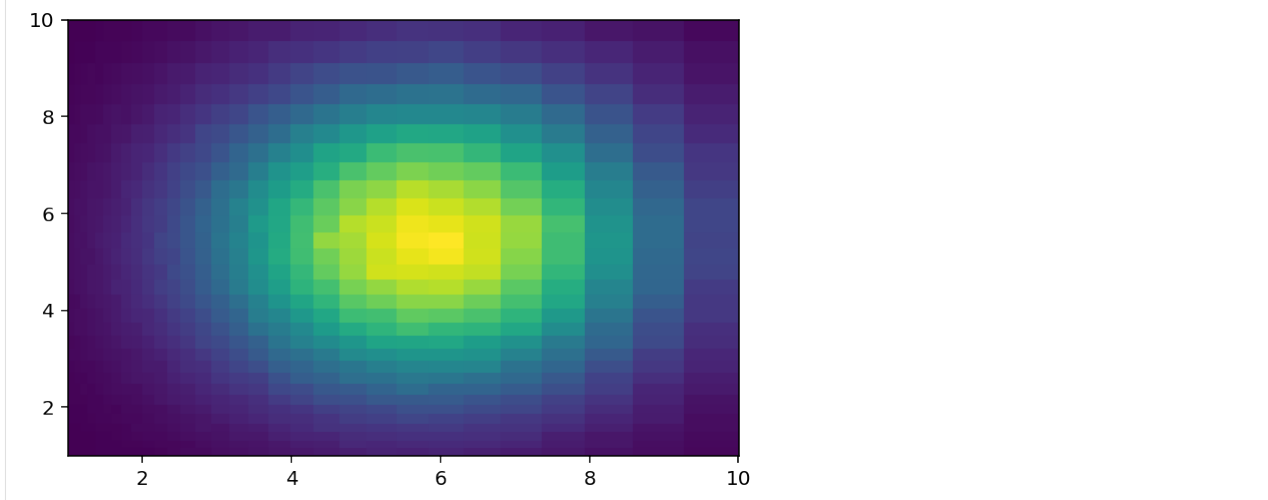
There are more axes types, and they all provide the same API in histograms, so they all just work without changes:

```
[44]: hist8 = bh.Histogram(
    bh.axis.Regular(30, 1, 10, transform=bh.axis.transform.log),
    bh.axis.Regular(30, 1, 10, transform=bh.axis.transform.sqrt),
)
```

```
[45]: hist8.reset()
hist8.fill(*make_2D_data(mean=(5, 5), widths=(5, 5)))
```

```
[45]: Histogram(
    Regular(30, 1, 10, transform=log),
    Regular(30, 1, 10, transform=sqrt),
    storage=Double()) # Sum: 903807.0 (1000000.0 with flow)
```

```
[46]: plothist2d(hist8);
```

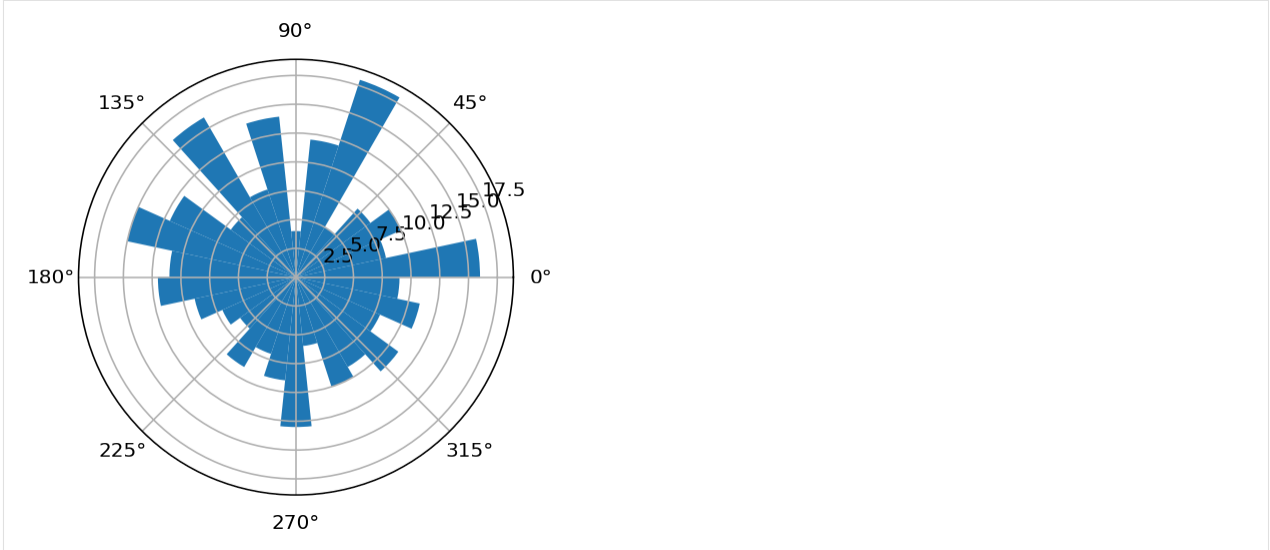


19.9 9: And, circular, too!

```
[47]: hist9 = bh.Histogram(bh.axis.Regular(30, 0, 2 * np.pi, circular=True))  
      hist9.fill(np.random.uniform(0, np.pi * 4, size=300))  
[47]: Histogram(Regular(30, 0, 6.28319, circular=True), storage=Double()) # Sum: 300.0
```

Now, the really complicated part, making a circular histogram:

```
[48]: ax = plt.subplot(111, polar=True)  
      plothist(hist9);
```



See the [Scikit-HEP Developer introduction](#) for a detailed description of best practices for developing Scikit-HEP packages.

CONTRIBUTING

20.1 Building from source

This repository has dependencies in submodules. Check out the repository like this:

```
git clone --recursive https://github.com/scikit-hep/boost-histogram.git
cd boost-histogram
```

```
git clone https://github.com/scikit-hep/boost-histogram.git
cd boost-histogram
git submodule update --init --depth 10
```

20.2 Setting up a development environment

20.2.1 Nox

The fastest way to start with development is to use nox. If you don't have nox, you can use `pipx run nox` to run it without installing, or `pipx install nox`. If you don't have pipx (pip for applications), then you can install with `pip install pipx` (the only case where installing an application with regular pip is reasonable). If you use macOS, then pipx and nox are both in brew, use `brew install pipx nox`.

To use, run nox. This will lint and test using every installed version of Python on your system, skipping ones that are not installed. You can also run specific jobs:

```
$ nox -l # List all the defined sessions
$ nox -s lint # Lint only
$ nox -s tests-3.9 # Python 3.9 tests only
$ nox -s docs -- serve # Build and serve the docs
$ nox -s make_pickle # Make a pickle file for this version
```

Nox handles everything for you, including setting up an temporary virtual environment for each run.

20.2.2 Pip

While developers often work in CMake, the “correct” way to develop a python package is in a virtual environment. This is how you would set one up with Python’s built-in venv:

```
python3 -m venv .env
source ./env/bin/activate
pip install -U pip
pip install -ve .[all]
```

You can set up a kernel for external Jupyter then deactivate your environment:

```
python -m ipykernel install --user --name boost-hist
deactivate
```

Now, you can run notebooks using your system JupyterLab, and it will list the environment as available!

To rebuild, rerun `pip install -ve .` from the environment, if the commit has changed, you will get a new build. Due to the `-e`, Python changes do not require a rebuild.

20.2.3 CMake

CMake is common for C++ development, and ties nicely to many C++ tools, like IDEs. If you want to use it for building, you can. Make a build directory and run CMake. If you have a specific Python you want to use, add `-DPYTHON_EXECUTABLE=$(which python)` or similar to the CMake line. If you need help installing the latest CMake version, [visit this page](#); one option is to use pip to install CMake.

Note: Since `setuptools` uses a subdirectory called `build`, it is *slightly* better to avoid making your CMake directory `build` as well. Also, you will often have multiple CMake directories (`build-release`, `build-debug`, etc.), so avoiding the descriptive name `build` is not a bad idea.

You have three options for running code in python:

1. Run from the build directory (only works with some commands, like `python -m pytest`, and not others, like `pytest`)
2. Add the build directory to your `PYTHONPATH` environment variable
3. Set `CMAKE_INSTALL_PREFIX` to your site-packages and install (recommended for virtual environments).

Here is the recommendation for a CMake install:

```
python3 -m venv env_cmake
source ./env_cmake/bin/activate
pip install -r dev-requirements.txt
cmake -S . -B build-debug \
    -GNinja \
    -DCMAKE_INSTALL_PREFIX=$(python -c "import distutils.sysconfig; print(distutils.
↪ sysconfig.get_python_lib(plat_specific=False, standard_lib=False))")
cmake --build build-debug -j4
cmake --install build-debug # Option 3 only
```

Note that option 3 will require reinstalling if the python files change, while options 1-2 will not if you have a recent version of CMake (symlinks are made).

This could be simplified if `pybind11` supported the new CMake `FindPython` tools.

20.3 Testing

Run the unit tests (requires pytest and NumPy).

```
python3 -m pytest
```

For CMake, you can also use the `test` target from anywhere, or use `python3 -m pytest` or `ctest` from the build directory.

The build requires `setuptools_scm`. The tests require `numpy`, `pytest`, and `pytest-benchmark`. `pytest-sugar` adds some nice formatting.

20.4 Benchmarking

You can enable benchmarking with `--benchmark-enable` when running tests. You can also run explicit performance tests with `scripts/performance_report.py`.

```
python3 -m pytest --benchmark-enable --benchmark-sort fullname
```

For example, if you want to benchmark before and after a change:

```
python3 -m pytest --benchmark-enable --benchmark-autosave
# Make change
python3 -m pytest --benchmark-enable --benchmark-autosave
pytest-benchmark compare 0001 0002 --sort fullname --histogram
```

Note, while the histogram option (`--histogram`) is nice, it does require `pygal` and `pygal.js` to be installed. Feel free to leave it off if not needed.

20.5 Formatting

Code should be well formatted; CI will check it and one of the authors can help reformat your code. If you want to check it yourself, you should use `pre-commit`.

Just `install pre-commit`, probably using `brew` on macOS or `pip` on other platforms, then run:

```
pre-commit install
```

Now all changed files will be checked every time you git commit. You can check it yourself (even without installing the hooks) using:

```
pre-commit run --all-files
```

We do not check `check-manifest` every time locally, since it is slow. You can trigger this manual check with:

```
pre-commit run --all-files --hook-stage manual check-manifest
```

Developers should update the pre-commit dependencies once in a while, you can do this automatically with:

```
pre-commit autoupdate
```

Note about skipping Docker

Pre-commit uses docker to ensure a consistent run of clang-format. If you do not want to install/run Docker, you should use `SKIP=docker-clang-format` when running pre-commit, and instead run `clang-format -style=file -i <files>` yourself.

20.6 Clang-Tidy

To run Clang tidy, the following recipe should work. Files will be modified in place, so you can use git to monitor the changes.

```
docker run --rm -v $PWD:/pybind11 -it silkeh/clang:10
apt-get update && apt-get install python3-dev
cmake -S pybind11/ -B build -DCMAKE_CXX_CLANG_TIDY="$(which clang-tidy);-fix"
cmake --build build
```

Remember to build single-threaded if applying fixes!

20.7 Include what you use

To run include what you use, install (`brew install include-what-you-use` on macOS), then run:

```
cmake -S . -B build-iwyu -DCMAKE_CXX_INCLUDE_WHAT_YOU_USE=$(which include-what-you-use)
cmake --build build
```

20.8 Timing steps

Make time/memory taken can be set `CMAKE_CXX_COMPILER_LAUNCHER/CMAKE_CXX_LINKER_LAUNCHER`. Some examples:

```
# Linux:
# "time"
# "time;-v"
# "time;-f;%U user %S system %E elapsed %P CPU %M KB"
# macOS:
# "time"
# macOS with brew install gnu-time:
# "gtime;-f;%U user %S system %E elapsed %P CPU %M KB"
#
```

20.9 Common tasks

This will checkout new versions of the dependencies. Example given using the fish shell.

```
for f in *
  cd $f
  git fetch
  git checkout boost-1.75.0 || echo "Not found"
  cd ..
end
```

- Finish merging open PRs that you want in the new version
- Add most recent changes to the docs/CHANGELOG.md
- Sync master with develop using `git checkout master; git merge develop --ff-only` and push
- Make sure the full wheel build runs on master without issues (manually trigger if needed)
- Make the GitHub release in the GitHub UI. Copy the changelog entries and links for that version; this has to be done as part of the release and tag procedure for archival tools (Zenodo) to pick them up correctly.
 - Title should be "Version <version number>"
 - Version tag should be "v" + major + "." + minor + "." + patch.
- GHA will build and send to PyPI for you when you release.
- Conda-forge will automatically make a PR to update within an hour or so, and it will merge automatically if it passes.

This requires LLVM 9+, and is based on [this post](#).

```
brew install llvm          # macOS way to get clang-9
python3 -m venv .env_core # general environment (no install will be made)
. .env_core/bin/activate
pip install -r dev-requirements.txt
CXX="/usr/local/opt/llvm/bin/clang++" cmake -S . -B build-llvm \
  -DCMAKE_CXX_FLAGS="-ftime-trace" \
  -DPYTHON_EXECUTABLE=$(which python)
cmake --build build-llvm/
```

Now open a browser with [SpeedScope](#), and load one of the files.

First, you need to install the [all contributor CLI](#):

```
yarn add --dev all-contributors-cli
```

Then, you can add contributors:

```
yarn all-contributors add henryiii maintenance,code,doc
```


SUPPORT

If you are stuck with a problem using Boost-histogram, please do get in touch at our [Issues](#) or [Gitter Channel](#). The developers are willing to help.

You can save time by following this procedure when reporting a problem:

- Do try to solve the problem on your own first. Read the documentation, including using the search feature, index and reference documentation.
- Search the issue archives to see if someone else already had the same problem.
- Before writing, try to create a minimal example that reproduces the problem. You'll get the fastest response if you can send just a handful of lines of code that show what isn't working.

CHANGELOG

22.1 Version 1.4

22.1.1 Version 1.4.1

Features

- NumPy 2 support. [#918](#)
- 32-bit Windows Python 3.12 wheel added (matching NumPy). [#920](#)

Bugfixes

- Support filling Integer axes with unsigned integers [#917](#)
- Avoid triggering NumPy 2 dev release install on Python 3.12. [#914](#)

Backend and docs

- Add missing API docs [#909](#)
- Use boost 1.84 [#920](#)

22.1.2 Version 1.4.0

Features

- `overflow=False` is now supported for `IntCategory` and `StrCategory`. [#883](#)

Changes

- Using `_storage_type` now produces a `DeprecationWarning` instead of `PendingDeprecationWarning`. [#801](#)
- Updated Boost to 1.82. The upper limit on Regular axes without overflow is now inclusive like NumPy. [#802](#)
- Produce more detailed error messages on C++ errors [#848](#)

Bugfixes

- Make filling an integer axis with a float array (also) an error. #876
- Include `-latomic` on `armv7l` #823

Backend and docs

- Add Python 3.12 support and binary wheels, also latest PyPy. `manylinux2014+` required. #880, #878
- Drop Python 3.6 support. #798
- Drop pre-built wheels for 32-bit Linux (NumPy also dropped). #849
- Add testing for WebAssembly (Pyodide). #850
- Use Ruff #829

22.2 Version 1.3

22.2.1 Version 1.3.2

Changes

- Added `storage_type` operator and `storage_type()` function #781, with pending deprecation for `_storage_type`. #786 #790
- Better errors generated for missing or incorrect sample to mean storage. #782
- Better error message when views are set with an incompatible array. #794

Bug fixes

- Patch broken sum with fully empty (0 bin) axis. #718
- Fix zero range `bh.numpy.histogram` to match `numpy.histogram` behavior. #721
- Avoid triggering `__init__` when copying (better support for subclasses with custom `init`'s). #759
- `IntCategory` now supports numbers larger than 2^{24} (now 2^{53}). #792
- Pick a subset now supported inside a larger expression. #793

Backend and docs

- Minor optimizations for UFuncs. #771
- Added Python 3.11 wheels. #789
- Include PyPy 3.9 binary wheels. #730
- Using `pybind11 2.10` #767
- Explicit `reset()` documentation. #783
- Minor cleanup and further removal of a little Python 2 back-compat code.
- Warnings have better `stacklevel` settings.

22.2.2 Version 1.3.1

Bug fixes

- Fixed regression with invalid `.project` input causing segfaults. #708
- Minor skips for specific tests on ppc64le, primarily for a NumPy bug. #707
- Avoid using EH for program control, better on Pyodide. #709
- Fix regression with exact float not being accepted for `.index` for `IntCategory` in 1.3.0. Add `hist` nox session to check downstream (manually for the moment). #710

22.2.3 Version 1.3.0

User changes

- PyPy 3.8 now supported with binary wheels. #677
- The GIL is released a little more often now. #662
- `AxesTuple` does not allow construction of non-axes. #680
- `KeyError` is now thrown when accessing a non-existent item in a `Category Axis` #689
- `WeightedViews` now support `np.cumsum` #699

Bug fixes

- Fixed `WeightedMean` storages producing NaN for `.variances()` #695
- Modify local include slightly to enable WebAssembly compilation in Pyodide #702

Developer changes

- Use PyLint in CI to check for some style issues #690
- Developer (CMake) installs no longer require toml #698

22.3 Version 1.2

22.3.1 Version 1.2.1

User changes

- musllinux wheels now provided along with manylinux #656

Bug fixes

- Fixed single-element negative growth fill [#654](#)

Developer changes

- No longer require Docker for clang-format, runs online too [#610](#)
- Using pybind11 2.8.0 [#658](#)

22.3.2 Version 1.2.0

User changes

- Python 3.10 officially supported, with wheels.
- Support subtraction on histograms [#636](#)
- Integer histograms are now signed [#636](#)

Bug fixes

- Support custom setters on AxesTuple subclasses. [#627](#)
- Faster picking if slices are not also used [#645](#) or if they are [#648](#) (1000x or more in some cases)
- Throw an error when an AxesTuple setter is the wrong length (inspired by zip strict in Python 3.10) [#627](#)
- Fix error thrown on comparison with axis and non-axis object [#631](#)
- Static typing no longer thinks `storage=` is required [#604](#)

Developer changes

- Support NumPy 1.21 for static type checking [#625](#)
- Use newer Boost 1.77 and Boost.Histogram 1.77+1 [#594](#)
- Provide nox support [#647](#)

22.4 Version 1.1

22.4.1 Version 1.1.0

User changes

- Experimentally support list selection on categorical axes [#577](#)
- Support Python 3.8 on Apple Silicon [#600](#)
- Scaling and addition with a scalar affect flow bins too [#580](#)
- Change `sum_of_deltas_squared` to `_sum_of_deltas_squared` (was an implementation detail) [#602](#)

Bug fixes

- Fix “picking” on a flow bin #576
- Better error message on getattr #596

Developer changes

- Test on Python 3.10 beta releases #600
- Provide a CMakeLists for quick standalone Boost.Histogram C++ experiments #591
- Adding logging with pytest failure output #575

22.5 Version 1.0

22.5.1 Version 1.0.2

- Fix scaling a weighted storage #559
- Fix partial summation over a Categorical axis #564
- Support running type checking from Python < 3.8 #542

22.5.2 Version 1.0.1

Subclassing Histogram changes

- A `family=` is no longer required if you *only* subclass Histogram. #533

Bug fixes

- Fix summing of Mean/WeightedMean accumulators #537
- Added missing dependency on `typing_extensions` for Python 3.6 & 3.7 #529

Typing changes

- Added Ellipsis support to typing. #525
- Better typing for Views. #530
- Fixed issue with Histogram copy constructor requiring metadata #532

22.5.3 Version 1.0.0

Dropped support for Python 2 and 3.5; removed large numbers of workarounds. Fully statically typed. API compatible with the final **0.x** release for most uses, except for subclassing; subclassing histogram components now uses Python 3 class keyword syntax to set families.

User changes

- Dropped Python 2.7 and 3.5 support [#512](#)
- Removed deprecated `.options` from axes. Use `.traits` instead. [#503](#)
- Full static typing available, UHI 0.1.2+ supported. [#516](#), [#517](#), [#519](#), [#520](#), [#521](#), [#523](#)

Subclassing Histogram changes

- Use keyword class family setting when subclassing histogram components instead of custom decorator. [#513](#)
- Structure of internal repr creation changed and made slightly more public. [#518](#)

Bug fixes

- Consistently show `metadata=` in repr if present; refactored internal repr handling [#518](#)
- Minor typing related fixes for rare bugs (especially in `numpy.py`, [#521](#))

22.6 Version 0.13

22.6.1 Version 0.13.2

- Backport fix scaling a weighted storage
- Backport fix partial summation over a Categorical axis

22.6.2 Version 0.13.1

- Backport fix for Mean/WeightedMean summing.
- Backport fix for `boost_histogram.numpy` density.
- Backport missing metadata from the repr's.
- Ignore `family=` on Histogram subclassing to make subclassing Histogram only possible in 1.x + 0.x code.

22.6.3 Version 0.13.0

PlottableProtocol provides a way to plot in different libraries, and easy access to common quantities. This is expected to be the final release for Python 2, and mostly equivalent in API to 1.0.

User changes

- Support for PlottableProtocol. You can now access `.values()`, `.counts()`, and `.variances()` on all storages; used by plotting libraries. `.kind` describes the Kind of the histogram (`bh.Kind.COUNT` or `bh.Kind.MEAN`). `.options` has been renamed to `.traits`, and a few more useful traits were added, like `.discrete`. Most other portions of the Protocol were already present. [#476](#)
- Removed deprecated `.rank` on histograms (since 0.8). Use `.ndim` instead. [#505](#)
- Supports converting user histogram objects that provide a `_to_boost_histogram_` method. [#483](#)
- A `view=True` parameter must now be passed to get a View instead of a standard NumPy values array from `to_numpy()`. [#498](#)

Bug fixes

- Added additional support for typing, fixing a couple of rare Python 2 bugs in the process [#493](#).
- The resulting histogram from `bh.numpy.*` functions is now reducible [#508](#)

Developer changes

- Use GitHub Actions for ARM compiling [#474](#)
- Apple Silicon support (since 0.12) [#495](#)
- Support compiling with C++17 [#502](#)
- Rename `NPY_NUM_BUILD_JOBS` to `CMAKE_BUILD_PARALLEL_LEVEL` for consistency with other Scikit-HEP projects. [#502](#)

22.7 Version 0.12

22.7.1 Version 0.12.0

Pressing forward to 1.0.

User changes

- You can now set all complex storages, either on a Histogram or a View with an (N+1)D array [#475](#)
- Axes are now normal `__dict__` classes, you can manipulate the `__dict__` as normal. Axes construction now lets you either use the old metadata shortcut or the `__dict__` inline. [#477](#)

Bug fixes

- Fixed slicing projection with one-sided slices [#479](#)
- Fixed issue if final bin of Variable histogram was infinite by updating to Boost 1.75 [#470](#)
- NumPy arrays can be used for weights in `bh.numpy` [#472](#)
- Vectorization for WeightedMean accumulators was broken [#475](#)

Developer changes

- Bumped to pybind11 version [#470](#)
- Black formatting used in notebooks too [#470](#)

Upgrade warning

If you are using `Axis.options`, please transition to `Axis.traits`. `traits` includes all the old options, along with some new traits, and matches the PlottableProtocol requirements.

22.8 Version 0.11

22.8.1 Version 0.11.1

Updating pybind11 to 2.6.0. [#443](#) Features:

- Python 3.9 support
- PyPy2 / PyPy3.6 / PyPy3.7 support
- Warnings on latest AppleClang fixed
- 40% faster accumulator fills, simpler implementation
- Segfaults when passing an object with a throwing repr fixed
- kwargs replaced older workarounds (partially at the moment)
- Using new `py::type` instead of `pybind11::detail` usage
- Enhanced CMake support, finds conda and venv now, uses `pybind11_find_import`
- Using setuptools support from pybind11 (previously vendored, so benefits have been available since 0.11.0)

Also cleans up SDists a bit. [#467](#)

22.8.2 Version 0.11.0

A release focused on preparing for the upcoming Hist 2.0 release.

User changes

- Arbitrary items can be set on an axis or histogram. [#450](#), [#456](#)
- Subclasses can customize the conversion procedure. [#456](#)

Bug fixes

- Fixed reading pickles from boost-histogram 0.6-0.8 [#445](#)
- Minor correctness fix [#446](#)
- Accidental install of typing on Python 3.5+ fixed
- Scalar ND fill fixed [#453](#)

Developer changes

- Updated to Boost 1.74 [#442](#)
- CMake installs version.py now too [#449](#)
- Updated setuptools infrastructure no longer requires NumPy [#451](#)
- Some basic clang-tidy checks are now being run [#455](#)

22.9 Version 0.10

22.9.1 Version 0.10.2

Quick fix for extra print statement in fill.

Bug fixes

- Fixed debugging print statement in fill. [#438](#)

Developer changes

- Added CI/pre-commit check for print statements [#438](#)
- Formatting CMakeLists now too [#439](#)

22.9.2 Version 0.10.1

Several fixes were made, mostly related to Weight storage histograms from Uproot 4.

Bug fixes

- Reduction on `h.axes.widths` supported again [#428](#)
- `WeightedSumView` supports standard array operations [#432](#)
- Operations shallow copy (non-copyable metadata supported) [#433](#)
- Pandas Series as samples/weights supported [#434](#)
- Support NumPy scalars in operations [#436](#)

22.9.3 Version 0.10.0

This version was released during PyHEP 2020. Several improvements were made to usability when plotting and indexing.

User changes

- `AxesTuple` array now support operations via `ArrayTuple` [#414](#)
- Support `sum` and `bh.rebin` without slice [#424](#)
- Nicer error messages in some cases [#415](#)
- Made a few properties hidden for accumulators that were not public [#418](#)
- Boolean now supports reduction, faster compile [#422](#)
- `AxesTuple` now available publicly for subprojects [#419](#)

Bug fixes

- Histograms support operations with arrays, no longer take the first element only [#417](#)

22.10 Version 0.9

22.10.1 Version 0.9.0

This version was released just before PyHEP 2020. Several important fixes were made, along with a few new features to better support downstream projects.

User changes

- metadata supported and propagated on Histograms (slots added) [#403](#)
- Added `dd=True` option in `to_numpy` [#406](#)
- Deprecated `cpp` module removed [#402](#)

Developer changes

- Subclasses can override axes generation [#401](#)
- `[dev]` extra now installs `pytest` [#401](#)

Bug fixes

- Fix `numpy.histogramdd` return structure [#406](#)
- Travis deploy multi-arch fixes [#399](#)
- Selecting on a bool axes supports 2D+ histograms [#398](#)
- Warnings fixed on NumPy 1.19+ [\[#404\]\[\[\]\]](#)

22.11 Version 0.8

22.11.1 Version 0.8.0

This version was released just before SciPy 2020 and Boost 1.74. Highlights include better accumulator views, simpler summing, better NumPy and Pandas compatibility, and sums on growing axes. Lots of backend work, including a new wheel building system, internal changes and better reliance on Boost.Histogram's C++ tools for actions like cropping.

User changes

- Weighted histogram cells can now be assigned directly from iterables [#375](#)
- Weighted views can be summed and added [#368](#)
- Sum is now identical to the built-in sum function [#365](#)
- Adding growing axis is better supported [#358](#)
- Slicing an `AxesTuple` now keeps the type [#384](#)
- `ndim` replaces `rank` for NumPy compatibility [#385](#)
- Any array-like supported in fill [#391](#), any iterable can be used for Categories [#392](#)
- Added Boolean axes, from Boost.Histogram 1.74 [#390](#)
- Division between histograms is supported [#393](#)
- More deprecated functionality removed

Bug fixes

- Support older versions of `CloudPickle` (issue also fixed upstream) [#343](#)
- Drop extra printout [#338](#)
- Throw an error instead of returning an incorrect result in more places [#386](#)

Developer changes

- Update Boost to 1.73 [#359](#), pybind11 to 2.5.0 [#351](#), Boost.Histogram to pre-1.74 [#388](#)
- Cropping no longer uses workaround [#373](#)
- Many more checks added to `pre-commit` [#366](#)
- Deprecating `cpp` interface [#391](#)
- Wheelbuilding migrated to `cibuildwheel` and GHA [#361](#)

22.12 Version 0.7

22.12.1 Version 0.7.0

This version removes deprecated functionality, and has several backend improvements. The most noticeable user-facing change is the multithreaded fill feature, which can enable significant speedups when you have a dataset that is much larger than the number of bins in your histogram and have free cores to use. Several small bugs have been fixed.

User changes

- Added `threads=` keyword to `.fill` and NumPy functions; 0 for automatic, default is 1 [#325](#)
- `.metadata` is now settable directly from the `AxesTuple` [#303](#)
- Deprecated items from 0.5.x now dropped [#301](#)
- `cpp` mode updates and fixes [#317](#)

Bug fixes

- Dict indexing is now identical to positional indexing, fixes “picking” axes in dict [#320](#)
- Passing `samples=None` is now always allowed in `.fill` [#325](#)

Developer changes

- Build system update, higher requirements for developers (only) [#314](#)
 - Version is now obtained from `setuptools_scm`, no longer stored in repo
- Removed `futures` requirement for Python 2 tests
- Updated Boost.Histogram, cleaner code with fewer workarounds

22.13 Version 0.6

22.13.1 Version 0.6.2

Common analysis tasks are now better supported. Much more complete documentation. Now using development branch of Boost.Histogram again.

Bug fixes

- Fix sum over category axes in indexing [#298](#)
- Allow single category item selection [#298](#)
- Allow slicing on axes without flow bins [#288](#), [#300](#)
- Sum repr no longer throws error [#293](#)

Developer changes

- Now using scikit-hep/azure-wheel-helpers via subtree [#292](#)

22.13.2 Version 0.6.1

Examples and notebooks are now up to date with the current state of the library. Using Boost 1.72 release.

User changes

- Slices and single values can be mixed in indexing [#279](#)
- UHI locators supported on axes [#280](#)

Bug fixes

- Properties on accumulator views now resolve correctly [#273](#)
- Division of a histogram by a number is supported again [#278](#)
- Setting a histogram with length one slice fixed [#279](#)
- NumPy functions now work with NumPy ints in `bins=` [#282](#)
- In-place addition avoids a copy [#284](#)

22.13.3 Version 0.6.0

This version fills out most of the remaining features missing from the 0.5.x series. You can now use all the storages without the original caveats; even the accumulators can be accessed array-at-a-time without copy, pickled quickly, and set array-at-a-time, as well.

The API has changed considerably, providing a more consistent experience in Python. Most of the classic API still works in this release, but will issue a warning and will be removed from the next release. Please use this release to transition existing 0.5.x code to the new API.

User changes

- Histogram and Axis classes now follow PEP 8 naming scheme (histogram->Histogram, regular->Regular, int->Int64 etc.) #192, #255
- You can now view a histogram with accumulators, with property access such as `h.view().value` #194
- Circular variable and integer axes added #231
- Split Category into `StrCategory` and `IntCategory`, now allows empty categories when `growth=True` #221
- `StrCategory` fills are safer and faster #239, #244
- Added axes transforms #192
- `Function(forward, inverse)` transform added, allowing ultra-fast C function pointer transforms #231
- You can now set histogram contents directly #250
- You can now sum over a range with endpoints #185
- `h.axes` now has the functions from axis as well. #183
- `bh.project` has become `bh.sum` #185
- `.reduce(...)` and the reducers in `bh.algorithm` have been removed in favor of dictionary based UHI slicing #259
- `bh.numpy` module interface updates, `histogram=bh.Histogram` replaces cryptic `bh=True`, and `density=True` is now supported in NumPy mode #256
- Added `hist.copy()` #218 and `hist.shape` #264
- Signatures are much nicer in Python 3 #188
- Reprs are better, various properties like `__module__` are now set correctly #200

Bug fixes

- Unlimited and AtomicInt storages now allow single item access #194
- `.view()` now no longer makes a copy #194
- Fixes related to string category axis fills #233, #230
- Axes are no longer copies, support setting metadata #238, #246
- Pickling accumulator storages is now comparable in performance simple storages #258

Developer changes

- The linux wheels are now 10-20x smaller [#229](#)
- The hist/axis classes are now pure Python, with a C++ object inside [#183](#)
- Most internal names changed, `core`->`_core`, etc. [#183](#)
- The `uhi` module is now `tag`. [#183](#)
- `boost_histogram.cpp` as `bh` provides C++ high-compatibility mode. [#183](#)
- Indexing tags now use full UHI instead of workarounds [#185](#)
- Removed log and sqrt special axes types [#231](#)
- Family and registration added, new casting system [#200](#)

22.14 Version 0.5

22.14.1 Version 0.5.2

User changes:

- `bh.loc` supports an offset [#164](#)
- Nicer reprs in several places [#167](#)
- Deprecate `.at` and `.axis` [#170](#)

Bug fixes:

- Use relative paths in `setup.py` to avoid resolving WSL paths on Windows [#162](#), [#163](#)
- Better `pybind11` support for Python 3.8 [#168](#)

Developer changes:

- Serialization code shared with `Boost.Histogram` [#166](#)
- Avoid unused PEP 517 isolation for now [#171](#) (may return with proper PEP 518 support eventually)

22.14.2 Version 0.5.1

User changes:

- Removed the `bh.indexed/h.indexed` iterator [#150](#)
- Added `.axes` `AxisTuple`, with direct access to properties [#150](#)
- Cleaned up tab completion in IPython [#150](#)

Bug fixes:

- Fixed a bug in the sdist missing `Boost.Variant2` [#154](#)
- Fixed filling on strided inputs [#158](#)

22.14.3 Version 0.5.0

First beta release and beginning of the changelog.

Known issues:

- Unlimited storage does not support pickling or classic multiprocessing
- Some non-simple storages do not support some forms of access, like `.view`
- Indexing and the array versions (such as centers) are incomplete and subject to change
- The numpy module is provisional and subject to change
- Docstrings and signatures will improve in later versions (especially on Python 3)

BOOST_HISTOGRAM

```
class boost_histogram._internal.hist.Histogram(*axes: Axis | CppAxis | Histogram | Any, storage:
                                             Storage = boost_histogram._core.storage.double,
                                             metadata: Any = None)
```

Bases: `object`

axes: `AxesTuple`

`copy(*, deep: bool = True) → H`

Make a copy of the histogram. Defaults to making a deep copy (axis metadata copied); use `deep=False` to avoid making a copy of axis metadata.

`counts(flow: bool = False) → np.typing.NDArray[Any]`

Returns the number of entries in each bin for an unweighted histogram or profile and an effective number of entries (defined below) for a weighted histogram or profile. An exotic generalized histogram could have no sensible `.counts`, so this is Optional and should be checked by Consumers.

If `kind == "MEAN"`, counts (effective or not) can and should be used to determine whether the mean value and its variance should be displayed (see documentation of values and variances, respectively). The counts should also be used to compute the error on the mean (see documentation of variances).

For a weighted histogram, counts is defined as $\text{sum_of_weights} ** 2 / \text{sum_of_weights_squared}$. It is equal or less than the number of times the bin was filled, the equality holds when all filled weights are equal. The larger the spread in weights, the smaller it is, but it is always 0 if filled 0 times, and 1 if filled once, and more than 1 otherwise.

Returns

`"np.typing.NDArray[Any]"[np.float64]`

`empty(flow: bool = False) → bool`

Check to see if the histogram has any non-default values. You can use `flow=True` to check flow bins too.

`fill(*args: Any | str, weight: Any | None = None, sample: Any | None = None, threads: int | None = None) → H`

Insert data into the histogram.

Parameters

- ***args** (`Union[Array[float], Array[int], Array[str], float, int, str]`) – Provide one value or array per dimension.
- **weight** (`List[Union[Array[float], Array[int], float, int, str]]`) – Provide weights (only if the histogram storage supports it)
- **sample** (`List[Union[Array[float], Array[int], Array[str], float, int, str]]`) – Provide samples (only if the histogram storage supports it)

- **threads** (*Optional* `[int]`) – Fill with threads. Defaults to None, which does not activate threaded filling. Using 0 will automatically pick the number of available threads (usually two per core).

property kind: `Kind`

Returns `Kind.COUNT` if this is a normal summing histogram, and `Kind.MEAN` if this is a mean histogram.

Returns

`Kind`

property ndim: `int`

Number of axes (dimensions) of the histogram.

project (**args:* `int`) → `H | float | Any`

Project to a single axis or several axes on a multidimensional histogram. Provided a list of axis numbers, this will produce the histogram over those axes only. Flow bins are used if available.

reset() → `H`

Clear the bin counters.

property shape: `tuple[int, ...]`

Tuple of axis sizes (not including underflow/overflow).

property size: `int`

Total number of bins in the histogram (including underflow/overflow).

property storage_type: `type[Storage]`

sum (*flow:* `bool = False`) → `float | Any`

Compute the sum over the histogram bins (optionally including the flow bins).

to_numpy (*flow:* `bool = False`, ***, *dd:* `bool = False`, *view:* `bool = False`) → `tuple[np.typing.NDArray[Any], ...]`
| `tuple[np.typing.NDArray[Any], tuple[np.typing.NDArray[Any], ...]]`

Convert to a NumPy style tuple of return arrays. Edges are converted to match NumPy standards, with upper edge inclusive, unlike boost-histogram, where upper edge is exclusive.

Parameters

- **flow** (*bool = False*) – Include the flow bins.
- **dd** (*bool = False*) – Use the histogramdd return syntax, where the edges are in a tuple. Otherwise, this is the histogram/histogram2d return style.
- **view** (*bool = False*) – The behavior for the return value. By default, this will return array of the values only regardless of the storage (which is all NumPy’s histogram function can do). `view=True` will return the boost-histogram view of the storage.

Returns

- **contents** (`Array[Any]`) – The bin contents
- ***edges** (`Array[float]`) – The edges for each dimension

values (*flow:* `bool = False`) → `np.typing.NDArray[Any]`

Returns the accumulated values. The counts for simple histograms, the sum of weights for weighted histograms, the mean for profiles, etc.

If counts is equal to 0, the value in that cell is undefined if `kind == “MEAN”`.

Parameters

flow – Enable flow bins. Not part of `PlottableHistogram`, but

included for consistency with other methods and flexibility.

Returns

`"np.typing.NDArray[Any]"[np.float64]`

variances(*flow*: *bool* = *False*) → `np.typing.NDArray[Any]` | `None`

Returns the estimated variance of the accumulated values. The sum of squared weights for weighted histograms, the variance of samples for profiles, etc. For an unweighed histogram where `kind == "COUNT"`, this should return the same as values if the histogram was not filled with weights, and `None` otherwise. If counts is equal to 1 or less, the variance in that cell is undefined if `kind == "MEAN"`. This must be written `<= 1`, and not `< 2`; when this effective counts (weighed mean), then counts could be less than 2 but more than 1.

If `kind == "MEAN"`, the counts can be used to compute the error on the mean as `sqrt(variances / counts)`, this works whether or not the entries are weighted if the weight variance was tracked by the implementation.

Currently, this always returns - but in the future, it will return `None` if a weighted fill is made on a unweighed storage.

Parameters

flow – Enable flow bins. Not part of `PlottableHistogram`, but

included for consistency with other methods and flexibility.

Returns

`"np.typing.NDArray[Any]"[np.float64]`

view(*flow*: *bool* = *False*) → `np.typing.NDArray[Any]` | `WeightedSumView` | `WeightedMeanView` | `MeanView`

Return a view into the data, optionally with overflow turned on.

BOOST_HISTOGRAM.AXIS

```
class boost_histogram.axis.ArrayTuple(iterable=(), /)
    Bases: tuple
    broadcast() → A
        The arrays in this tuple will be compressed if possible to save memory. Use this method to broadcast them
        out into their full memory representation.

class boost_histogram.axis.AxesTuple(_AxesTuple__iterable: Iterable[Axis])
    Bases: tuple
    bin(*indexes: float) → tuple[float, ...]
        Return the edges of the bins as a tuple for a continuous axis or the bin value for a non-continuous axis,
        when given an index.

    property centers: ArrayTuple

    property edges: ArrayTuple

    property extent: tuple[int, ...]

    index(*values: float) → tuple[float, ...]
        Return the fractional index(es) given a value (or values) on the axis.

    property size: tuple[int, ...]

    value(*indexes: float) → tuple[float, ...]
        Return the value(s) given an (fractional) index (or indices).

    property widths: ArrayTuple

class boost_histogram.axis.Axis(ax: Any, metadata: dict[str, Any] | None, __dict__: dict[str, Any] | None)
    Bases: object
    bin(index: float) → int | str | tuple[float, float]
        Return the edges of the bins as a tuple for a continuous axis or the bin value for a non-continuous axis,
        when given an index.

    property centers: np.typing.NDArray[Any]
        An array of bin centers.

    property edges: np.typing.NDArray[Any]

    property extent: int
        Return number of bins including under- and overflow.
```

index(value: *float* | *str*) → *int*

Return the fractional index(es) given a value (or values) on the axis.

property size: *int*

Return number of bins excluding under- and overflow.

property traits: *Traits*

Get traits for the axis - read only properties of a specific axis.

value(index: *float*) → *float*

Return the value(s) given an (fractional) index (or indices).

property widths: *np.typing.NDArray[Any]*

An array of bin widths.

class boost_histogram.axis.**Boolean**(**metadata: Any = None*, *__dict__*: *dict[str, Any] | None = None*)

Bases: *Axis*

class boost_histogram.axis.**IntCategory**(*categories: Iterable[int]*, *, *metadata: Any = None*, *growth: bool = False*, *overflow: bool = True*, *__dict__*: *dict[str, Any] | None = None*)

Bases: *BaseCategory*

class boost_histogram.axis.**Integer**(*start: int*, *stop: int*, *, *metadata: Any = None*, *underflow: bool = True*, *overflow: bool = True*, *growth: bool = False*, *circular: bool = False*, *__dict__*: *dict[str, Any] | None = None*)

Bases: *Axis*

class boost_histogram.axis.**Regular**(*bins: int*, *start: float*, *stop: float*, *, *metadata: Any = None*, *underflow: bool = True*, *overflow: bool = True*, *growth: bool = False*, *circular: bool = False*, *transform: AxisTransform | None = None*, *__dict__*: *dict[str, Any] | None = None*)

Bases: *Axis*

property transform: *AxisTransform | None*

class boost_histogram.axis.**StrCategory**(*categories: Iterable[str]*, *, *metadata: Any = None*, *growth: bool = False*, *overflow: bool = True*, *__dict__*: *dict[str, Any] | None = None*)

Bases: *BaseCategory*

index(value: *float* | *str*) → *int*

Return the fractional index(es) given a value (or values) on the axis.

class boost_histogram.axis.**Traits**(*underflow: 'bool' = False*, *overflow: 'bool' = False*, *circular: 'bool' = False*, *growth: 'bool' = False*, *continuous: 'bool' = False*, *ordered: 'bool' = False*)

Bases: *object*

circular: *bool* = *False*

continuous: *bool* = *False*

property discrete: *bool*

True if axis is not continuous

growth: *bool* = *False*

ordered: `bool` = `False`

overflow: `bool` = `False`

underflow: `bool` = `False`

class `boost_histogram.axis.Variable`(*edges: Iterable[float], *, metadata: Any = None, underflow: bool = True, overflow: bool = True, growth: bool = False, circular: bool = False, __dict__: dict[str, Any] | None = None*)

Bases: `Axis`

BOOST_HISTOGRAM.AXIS.TRANSFORM

```
class boost_histogram.axis.transform.AxisTransform
```

Bases: `object`

```
    forward(value: float) → float
```

Compute the forward transform

```
    inverse(value: float) → float
```

Compute the inverse transform

```
class boost_histogram.axis.transform.Function(forward: Any, inverse: Any, *, convert: Any = None,
                                              name: str = "")
```

Bases: `AxisTransform`

```
class boost_histogram.axis.transform.Pow(power: float)
```

Bases: `AxisTransform`

```
    property power: float
```

The power of the transform

BOOST_HISTOGRAM.ACCUMULATORS

`boost_histogram.accumulators.Accumulator`

alias of [Any](#)

BOOST_HISTOGRAM.NUMPY

```
boost_histogram.numpy.histogram(a: ArrayLike, bins: int | str | np.typing.NDArray[Any] = 10, range:
                                tuple[float, float] | None = None, normed: None = None, weights:
                                ArrayLike | None = None, density: bool = False, *, histogram: None |
                                type[_hist.Histogram] = None, storage: _storage.Storage | None = None,
                                threads: int | None = None) → Any
```

Return a boost-histogram object using the same arguments as numpy's histogram. This does not support the deprecated `normed=True` argument. Three extra arguments are added: `histogram=bh.Histogram` will enable object based output, `storage=bh.storage.*` lets you set the storage used, and `threads=int` lets you set the number of threads to fill with (0 for auto, None for 1).

Compute the histogram of a dataset.

Parameters

- **a** (*array_like*) – Input data. The histogram is computed over the flattened array.
- **bins** (*int or sequence of scalars or str, optional*) – If *bins* is an int, it defines the number of equal-width bins in the given range (10, by default). If *bins* is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge, allowing for non-uniform bin widths.

Added in version 1.11.0.

If *bins* is a string, it defines the method used to calculate the optimal bin width, as defined by *histogram_bin_edges*.

- **range** (*(float, float), optional*) – The lower and upper range of the bins. If not provided, range is simply `(a.min(), a.max())`. Values outside the range are ignored. The first element of the range must be less than or equal to the second. *range* affects the automatic bin computation as well. While bin width is computed to be optimal based on the actual data within *range*, the bin count will fill the entire range including portions containing no data.
- **weights** (*array_like, optional*) – An array of weights, of the same shape as *a*. Each value in *a* only contributes its associated weight towards the bin count (instead of 1). If *density* is True, the weights are normalized, so that the integral of the density over the range remains 1.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability *density* function at the bin, normalized such that the *integral* over the range is 1. Note that the sum of the histogram values will not be equal to 1 unless bins of unity width are chosen; it is not a probability *mass* function.

Returns

- **hist** (*array*) – The values of the histogram. See *density* and *weights* for a description of the possible semantics.
- **bin_edges** (*array of dtype float*) – Return the bin edges (`length(hist)+1`).

See also:

[`histogramdd`](#), [`bincount`](#), [`searchsorted`](#), [`digitize`](#), [`histogram_bin_edges`](#)

Notes

All but the last (righthand-most) bin is half-open. In other words, if *bins* is:

```
[1, 2, 3, 4]
```

then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

Examples

```
>>> np.histogram([1, 2, 1], bins=[0, 1, 2, 3])
(array([0, 2, 1]), array([0, 1, 2, 3]))
>>> np.histogram(np.arange(4), bins=np.arange(5), density=True)
(array([0.25, 0.25, 0.25, 0.25]), array([0, 1, 2, 3, 4]))
>>> np.histogram([1, 2, 1], [1, 0, 1], bins=[0,1,2,3])
(array([1, 4, 1]), array([0, 1, 2, 3]))
```

```
>>> a = np.arange(5)
>>> hist, bin_edges = np.histogram(a, density=True)
>>> hist
array([0.5, 0. , 0.5, 0. , 0. , 0.5, 0. , 0.5, 0. , 0.5])
>>> hist.sum()
2.4999999999999996
>>> np.sum(hist * np.diff(bin_edges))
1.0
```

Added in version 1.11.0.

Automated Bin Selection Methods example, using 2 peak random data with 2000 points:

```
>>> import matplotlib.pyplot as plt
>>> rng = np.random.RandomState(10) # deterministic random data
>>> a = np.hstack((rng.normal(size=1000),
...               rng.normal(loc=5, scale=2, size=1000)))
>>> _ = plt.hist(a, bins='auto') # arguments are passed to np.histogram
>>> plt.title("Histogram with 'auto' bins")
Text(0.5, 1.0, "Histogram with 'auto' bins")
>>> plt.show()
```

`boost_histogram.numpy.histogram2d`(*x: object, y: object, bins: int | tuple[int, int] = 10, range: None | Sequence[None | tuple[float, float]] = None, normed: None = None, weights: object | None = None, density: bool = False, *, histogram: None | type[Histogram] = None, storage: Storage = boost_histogram._core.storage.double, threads: int | None = None*) → Any

Return a boost-histogram object using the same arguments as numpy's `histogram2d`. This does not support the deprecated `normed=True` argument. Three extra arguments are added: `histogram=bh.Histogram` will enable object based output, `storage=bh.storage.*` lets you set the storage used, and `threads=int` lets you set the number of threads to fill with (0 for auto, None for 1).

Compute the bi-dimensional histogram of two data samples.

Parameters

- **x** (*array_like*, *shape* (N,)) – An array containing the x coordinates of the points to be histogrammed.
- **y** (*array_like*, *shape* (N,)) – An array containing the y coordinates of the points to be histogrammed.
- **bins** (*int* or *array_like* or [*int*, *int*] or [*array*, *array*], *optional*) – The bin specification:
 - If *int*, the number of bins for the two dimensions (`nx=ny=bins`).
 - If *array_like*, the bin edges for the two dimensions (`x_edges=y_edges=bins`).
 - If [*int*, *int*], the number of bins in each dimension (`nx, ny = bins`).
 - If [*array*, *array*], the bin edges in each dimension (`x_edges, y_edges = bins`).
 - A combination [*int*, *array*] or [*array*, *int*], where *int* is the number of bins and *array* is the bin edges.
- **range** (*array_like*, *shape*(2,2), *optional*) – The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the *bins* parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.
- **density** (*bool*, *optional*) – If False, the default, returns the number of samples in each bin. If True, returns the probability *density* function at the bin, `bin_count / sample_count / bin_area`.
- **weights** (*array_like*, *shape*(N,), *optional*) – An array of values `w_i` weighing each sample (`x_i, y_i`). Weights are normalized to 1 if *density* is True. If *density* is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

Returns

- **H** (*ndarray*, *shape*(`nx, ny`)) – The bi-dimensional histogram of samples *x* and *y*. Values in *x* are histogrammed along the first dimension and values in *y* are histogrammed along the second dimension.
- **xedges** (*ndarray*, *shape*(`nx+1`,)) – The bin edges along the first dimension.
- **yedges** (*ndarray*, *shape*(`ny+1`,)) – The bin edges along the second dimension.

See also:

[*histogram*](#)

1D histogram

[*histogramdd*](#)

Multidimensional histogram

Notes

When *density* is True, then the returned histogram is the sample density, defined such that the sum over bins of the product `bin_value * bin_area` is 1.

Please note that the histogram does not follow the Cartesian convention where *x* values are on the abscissa and *y* values on the ordinate axis. Rather, *x* is histogrammed along the first dimension of the array (vertical), and *y* along the second dimension of the array (horizontal). This ensures compatibility with *histogramdd*.

Examples

```
>>> from matplotlib.image import NonUniformImage
>>> import matplotlib.pyplot as plt
```

Construct a 2-D histogram with variable bin width. First define the bin edges:

```
>>> xedges = [0, 1, 3, 5]
>>> yedges = [0, 2, 3, 4, 6]
```

Next we create a histogram *H* with random bin content:

```
>>> x = np.random.normal(2, 1, 100)
>>> y = np.random.normal(1, 1, 100)
>>> H, xedges, yedges = np.histogram2d(x, y, bins=(xedges, yedges))
>>> # Histogram does not follow Cartesian convention (see Notes),
>>> # therefore transpose H for visualization purposes.
>>> H = H.T
```

`imshow` can only display square bins:

```
>>> fig = plt.figure(figsize=(7, 3))
>>> ax = fig.add_subplot(131, title='imshow: square bins')
>>> plt.imshow(H, interpolation='nearest', origin='lower',
...           extent=[xedges[0], xedges[-1], yedges[0], yedges[-1]])
<matplotlib.image.AxesImage object at 0x...>
```

`pcolormesh` can display actual edges:

```
>>> ax = fig.add_subplot(132, title='pcolormesh: actual edges',
...                       aspect='equal')
>>> X, Y = np.meshgrid(xedges, yedges)
>>> ax.pcolormesh(X, Y, H)
<matplotlib.collections.QuadMesh object at 0x...>
```

`NonUniformImage` can be used to display actual bin edges with interpolation:

```
>>> ax = fig.add_subplot(133, title='NonUniformImage: interpolated',
...                       aspect='equal', xlim=xedges[[0, -1]], ylim=yedges[[0, -1]])
>>> im = NonUniformImage(ax, interpolation='bilinear')
>>> xcenters = (xedges[:-1] + xedges[1:]) / 2
>>> ycenters = (yedges[:-1] + yedges[1:]) / 2
>>> im.set_data(xcenters, ycenters, H)
>>> ax.add_image(im)
>>> plt.show()
```

It is also possible to construct a 2-D histogram without specifying bin edges:

```
>>> # Generate non-symmetric test data
>>> n = 10000
>>> x = np.linspace(1, 100, n)
>>> y = 2*np.log(x) + np.random.rand(n) - 0.5
>>> # Compute 2d histogram. Note the order of x/y and xedges/yedges
>>> H, yedges, xedges = np.histogram2d(y, x, bins=20)
```

Now we can plot the histogram using `pcolormesh`, and a `hexbin` for comparison.

```
>>> # Plot histogram using pcolormesh
>>> fig, (ax1, ax2) = plt.subplots(ncols=2, sharey=True)
>>> ax1.pcolormesh(xedges, yedges, H, cmap='rainbow')
>>> ax1.plot(x, 2*np.log(x), 'k-')
>>> ax1.set_xlim(x.min(), x.max())
>>> ax1.set_ylim(y.min(), y.max())
>>> ax1.set_xlabel('x')
>>> ax1.set_ylabel('y')
>>> ax1.set_title('histogram2d')
>>> ax1.grid()
```

```
>>> # Create hexbin plot for comparison
>>> ax2.hexbin(x, y, gridsize=20, cmap='rainbow')
>>> ax2.plot(x, 2*np.log(x), 'k-')
>>> ax2.set_title('hexbin')
>>> ax2.set_xlim(x.min(), x.max())
>>> ax2.set_xlabel('x')
>>> ax2.grid()
```

```
>>> plt.show()
```

`boost_histogram.numpy.histogramdd(a: tuple[ArrayLike, ...], bins: int | tuple[int, ...] | tuple[np.typing.NDArray[Any], ...] = 10, range: None | Sequence[None | tuple[float, float]] = None, normed: None = None, weights: ArrayLike | None = None, density: bool = False, *, histogram: None | type[_hist.Histogram] = None, storage: _storage.Storage = boost_histogram._core.storage.double, threads: int | None = None) → Any`

Return a boost-histogram object using the same arguments as numpy's `histogramdd`. This does not support the deprecated `normed=True` argument. Three extra arguments are added: `histogram=bh.Histogram` will enable object based output, `storage=bh.storage.*` lets you set the storage used, and `threads=int` lets you set the number of threads to fill with (0 for auto, None for 1).

Compute the multidimensional histogram of some data.

Parameters

- **sample** ((*N*, *D*) array, or (*N*, *D*) array_like) – The data to be histogrammed.

Note the unusual interpretation of `sample` when an `array_like`:

- When an array, each row is a coordinate in a *D*-dimensional space - such as `histogramdd(np.array([p1, p2, p3]))`.
- When an `array_like`, each element is the list of values for single coordinate - such as `histogramdd((X, Y, Z))`.

The first form should be preferred.

- **bins** (*sequence or int, optional*) – The bin specification:
 - A sequence of arrays describing the monotonically increasing bin edges along each dimension.
 - The number of bins for each dimension (`nx, ny, ... =bins`)
 - The number of bins for all dimensions (`nx=ny=...=bins`).
- **range** (*sequence, optional*) – A sequence of length `D`, each an optional (lower, upper) tuple giving the outer bin edges to be used if the edges are not given explicitly in *bins*. An entry of `None` in the sequence results in the minimum and maximum values being used for the corresponding dimension. The default, `None`, is equivalent to passing a tuple of `D` `None` values.
- **density** (*bool, optional*) – If `False`, the default, returns the number of samples in each bin. If `True`, returns the probability *density* function at the bin, `bin_count / sample_count / bin_volume`.
- **weights** (*(N,) array_like, optional*) – An array of values w_i weighing each sample (x_i, y_i, z_i, \dots). Weights are normalized to 1 if density is `True`. If density is `False`, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

Returns

- **H** (*ndarray*) – The multidimensional histogram of sample `x`. See density and weights for the different possible semantics.
- **edges** (*list*) – A list of `D` arrays describing the bin edges for each dimension.

See also:

[*histogram*](#)

1-D histogram

[*histogram2d*](#)

2-D histogram

Examples

```
>>> r = np.random.randn(100,3)
>>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
>>> H.shape, edges[0].size, edges[1].size, edges[2].size
((5, 8, 4), 6, 9, 5)
```

BOOST_HISTOGRAM.STORAGE

```
class boost_histogram.storage.AtomicInt64(*args: Any, **kwargs: Any)
    Bases: atomic_int64, Storage
    accumulator
        alias of int

class boost_histogram.storage.Double(*args: Any, **kwargs: Any)
    Bases: double, Storage
    accumulator
        alias of float

class boost_histogram.storage.Int64(*args: Any, **kwargs: Any)
    Bases: int64, Storage
    accumulator
        alias of int

class boost_histogram.storage.Mean(*args: Any, **kwargs: Any)
    Bases: mean, Storage

class boost_histogram.storage.Storage
    Bases: object
    accumulator: ClassVar[type[int] | type[float] |
type[boost_histogram._core.accumulators.WeightedMean] |
type[boost_histogram._core.accumulators.WeightedSum] |
type[boost_histogram._core.accumulators.Mean]]

class boost_histogram.storage.Unlimited(*args: Any, **kwargs: Any)
    Bases: unlimited, Storage
    accumulator
        alias of float

class boost_histogram.storage.Weight(*args: Any, **kwargs: Any)
    Bases: weight, Storage

class boost_histogram.storage.WeightedMean(*args: Any, **kwargs: Any)
    Bases: weighted_mean, Storage
```


BOOST_HISTOGRAM.TAG

```
class boost_histogram.tag.Locator(offset: int = 0)
```

Bases: `object`

NAME = ''

offset

```
class boost_histogram.tag.Slicer
```

Bases: `object`

This is a simple class to make slicing inside dictionaries simpler. This is how it should be used:

```
s = bh.tag.Slicer()
```

```
h[{0: s[:,bh.rebin(2)]]} # rebin axis 0 by two
```

```
class boost_histogram.tag.at(value: int)
```

Bases: `object`

value

```
class boost_histogram.tag.loc(value: str | float, offset: int = 0)
```

Bases: `Locator`

value

```
class boost_histogram.tag.rebin(value: int)
```

Bases: `object`

factor

```
boost_histogram.tag.sum(iterable, /, start=0)
```

Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

BOOST_HISTOGRAM.VERSION

ACKNOWLEDGEMENTS

This library was primarily developed by Henry Schreiner and Hans Dembinski.

Support for this work was provided by the National Science Foundation cooperative agreement OAC-1836650 (IRIS-HEP) and OAC-1450377 (DIANA/HEP). Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

b

- `boost_histogram._internal.hist`, 95
- `boost_histogram.accumulators`, 105
- `boost_histogram.axis`, 99
- `boost_histogram.axis.transform`, 103
- `boost_histogram.numpy`, 107
- `boost_histogram.storage`, 113
- `boost_histogram.tag`, 115

A

accumulator (*boost_histogram.storage.AtomicInt64* attribute), 113
 accumulator (*boost_histogram.storage.Double* attribute), 113
 accumulator (*boost_histogram.storage.Int64* attribute), 113
 accumulator (*boost_histogram.storage.Storage* attribute), 113
 accumulator (*boost_histogram.storage.Unlimited* attribute), 113
 Accumulator (in *boost_histogram.accumulators*), 105
 ArrayTuple (class in *boost_histogram.axis*), 99
 at (class in *boost_histogram.tag*), 115
 AtomicInt64 (class in *boost_histogram.storage*), 113
 axes (*boost_histogram._internal.hist.Histogram* attribute), 95
 AxesTuple (class in *boost_histogram.axis*), 99
 Axis (class in *boost_histogram.axis*), 99
 AxisTransform (class in *boost_histogram.axis.transform*), 103

B

bin() (*boost_histogram.axis.AxesTuple* method), 99
 bin() (*boost_histogram.axis.Axis* method), 99
 Boolean (class in *boost_histogram.axis*), 100
 boost_histogram._internal.hist module, 95
 boost_histogram.accumulators module, 105
 boost_histogram.axis module, 99
 boost_histogram.axis.transform module, 103
 boost_histogram.numpy module, 107
 boost_histogram.storage module, 113
 boost_histogram.tag module, 115

broadcast() (*boost_histogram.axis.ArrayTuple* method), 99

C

centers (*boost_histogram.axis.AxesTuple* property), 99
 centers (*boost_histogram.axis.Axis* property), 99
 circular (*boost_histogram.axis.Traits* attribute), 100
 continuous (*boost_histogram.axis.Traits* attribute), 100
 copy() (*boost_histogram._internal.hist.Histogram* method), 95
 counts() (*boost_histogram._internal.hist.Histogram* method), 95

D

discrete (*boost_histogram.axis.Traits* property), 100
 Double (class in *boost_histogram.storage*), 113

E

edges (*boost_histogram.axis.AxesTuple* property), 99
 edges (*boost_histogram.axis.Axis* property), 99
 empty() (*boost_histogram._internal.hist.Histogram* method), 95
 extent (*boost_histogram.axis.AxesTuple* property), 99
 extent (*boost_histogram.axis.Axis* property), 99

F

factor (*boost_histogram.tag.rebin* attribute), 115
 fill() (*boost_histogram._internal.hist.Histogram* method), 95
 forward() (*boost_histogram.axis.transform.AxisTransform* method), 103
 Function (class in *boost_histogram.axis.transform*), 103

G

growth (*boost_histogram.axis.Traits* attribute), 100

H

Histogram (class in *boost_histogram._internal.hist*), 95
 histogram() (in module *boost_histogram.numpy*), 107
 histogram2d() (in module *boost_histogram.numpy*), 108

histogramdd() (in module boost_histogram.numpy), 111

I

index() (boost_histogram.axis.AxesTuple method), 99
 index() (boost_histogram.axis.Axis method), 99
 index() (boost_histogram.axis.StrCategory method), 100
 Int64 (class in boost_histogram.storage), 113
 IntCategory (class in boost_histogram.axis), 100
 Integer (class in boost_histogram.axis), 100
 inverse() (boost_histogram.axis.transform.AxisTransform method), 103

K

kind (boost_histogram._internal.hist.Histogram property), 96

L

loc (class in boost_histogram.tag), 115
 Locator (class in boost_histogram.tag), 115

M

Mean (class in boost_histogram.storage), 113
 module
 boost_histogram._internal.hist, 95
 boost_histogram.accumulators, 105
 boost_histogram.axis, 99
 boost_histogram.axis.transform, 103
 boost_histogram.numpy, 107
 boost_histogram.storage, 113
 boost_histogram.tag, 115

N

NAME (boost_histogram.tag.Locator attribute), 115
 ndim (boost_histogram._internal.hist.Histogram property), 96

O

offset (boost_histogram.tag.Locator attribute), 115
 ordered (boost_histogram.axis.Traits attribute), 100
 overflow (boost_histogram.axis.Traits attribute), 101

P

Pow (class in boost_histogram.axis.transform), 103
 power (boost_histogram.axis.transform.Pow property), 103
 project() (boost_histogram._internal.hist.Histogram method), 96

R

rebin (class in boost_histogram.tag), 115
 Regular (class in boost_histogram.axis), 100

reset() (boost_histogram._internal.hist.Histogram method), 96

S

shape (boost_histogram._internal.hist.Histogram property), 96
 size (boost_histogram._internal.hist.Histogram property), 96
 size (boost_histogram.axis.AxesTuple property), 99
 size (boost_histogram.axis.Axis property), 100
 Slicer (class in boost_histogram.tag), 115
 Storage (class in boost_histogram.storage), 113
 storage_type (boost_histogram._internal.hist.Histogram property), 96
 StrCategory (class in boost_histogram.axis), 100
 sum() (boost_histogram._internal.hist.Histogram method), 96
 sum() (in module boost_histogram.tag), 115

T

to_numpy() (boost_histogram._internal.hist.Histogram method), 96
 traits (boost_histogram.axis.Axis property), 100
 Traits (class in boost_histogram.axis), 100
 transform (boost_histogram.axis.Regular property), 100

U

underflow (boost_histogram.axis.Traits attribute), 101
 Unlimited (class in boost_histogram.storage), 113

V

value (boost_histogram.tag.at attribute), 115
 value (boost_histogram.tag.loc attribute), 115
 value() (boost_histogram.axis.AxesTuple method), 99
 value() (boost_histogram.axis.Axis method), 100
 values() (boost_histogram._internal.hist.Histogram method), 96
 Variable (class in boost_histogram.axis), 101
 variances() (boost_histogram._internal.hist.Histogram method), 97
 view() (boost_histogram._internal.hist.Histogram method), 97

W

Weight (class in boost_histogram.storage), 113
 WeightedMean (class in boost_histogram.storage), 113
 widths (boost_histogram.axis.AxesTuple property), 99
 widths (boost_histogram.axis.Axis property), 100