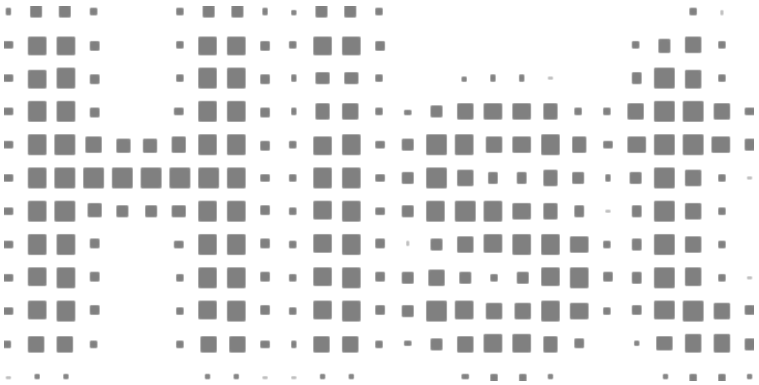
Hist

Henry Schreiner and Nino Lau

Sep 12, 2023

USER GUIDE



CHAPTER

ONE

INTRODUCTION

Hist is a powerful Histogramming tool for analysis based on boost-histogram (the Python binding of the Histogram library in Boost). It is a friendly analysis-focused project that uses boost-histogram as a backend to do the work, but provides plotting tools, shortcuts, and new ideas.

To get an idea of creating histograms in Hist looks like, you can take a look at the *Examples*. Once you have a feel for what is involved in using Hist, we recommend you start by following the instructions in *Installation*. Then, go through the User Guide starting with *Quickstart*, and read the *Reference* documentation. We value your contributions and you can follow the instructions in *Contributing*. Finally, if you're having problems, please do let us know at our *Support* page.

1.1 Installation

Hist is available on PyPI. You can install this library from PyPI with pip:

```
python3 -m pip install "hist[plot]"
```

If you do not need the plotting features, you can skip the `[plot]` extra.

You can also install it with Conda from conda-forge.

```
conda install -c conda-forge hist
```

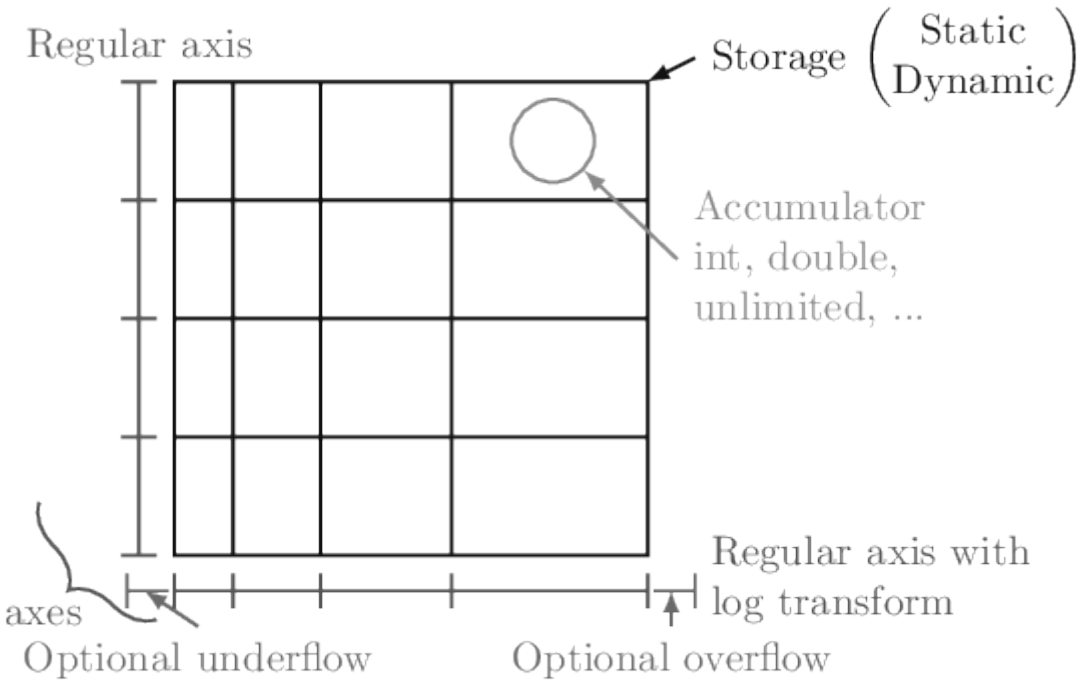
Supported platforms are identical to boost-histogram, though for the latest version of Hist you need Python 3.7+.

1.2 Quickstart

All of the examples will assume the following import:

```
import hist
from hist import Hist
```

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



1.2.1 Making a histogram

You can make a histogram like this:

```
h = Hist(hist.axis.Regular(bins=10, start=0, stop=1, name="x"))
```

If you'd like to type less, you can leave out the keywords for the numbers.

```
h = Hist(hist.axis.Regular(10, 0, 1, name="x"))
```

Hist also supports a "quick-construct" system, which does not require using anything beyond the `Hist` class:

```
h = Hist.new.Regular(10, 0, 1, name="x").Double()
```

Note that you have to specify the storage at the end (but this does make it easier to use `Weight` or other useful storages).

The exact same syntax is used any number of dimensions:

```
hist3D = (
    Hist.new.Regular(10, 0, 100, circular=True, name="x")
    .Regular(10, 0.0, 10.0, name="y")
    .Variable([1, 2, 3, 4, 5, 5.5, 6], name="z")
    .Weight()
)
```

See *Axes* and *Transform*.

You can also select a different storage with the `storage=` keyword argument; see *Storages* for details about the other storages.

1.2.2 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:

```
h = Hist(hist.axis.Regular(10, 0.0, 1.0, name="x"))
h.fill(0.9)
h.fill([0.9, 0.3, 0.4])
```

1.2.3 Slicing and rebinning

You can slice into a histogram using bin coordinates or data coordinates by appending a `j` to an index. You can also rebin with a number ending in `j` in the third slice entry, or remove an entire axis using `sum`:

```
h = Hist(
    hist.axis.Regular(10, 0, 1, name="x"),
    hist.axis.Regular(10, 0, 1, name="y"),
    hist.axis.Regular(10, 0, 1, name="z"),
)
mini = h[1:5, 0.2j:0.9j, sum]
# Will be 4 bins x 7 bins
```

See *Indexing*.

1.2.4 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 *Plots*.

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 = h.view()
```

1.2.5 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:

```
h[2] = 3.5
h[hist.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:

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

See *Indexing*.

1.2.6 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 = h.axes[0]
X, Y = h.axes.centers
```

See *Axes*.

1.2.7 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.

1.2.8 Computing with Histograms

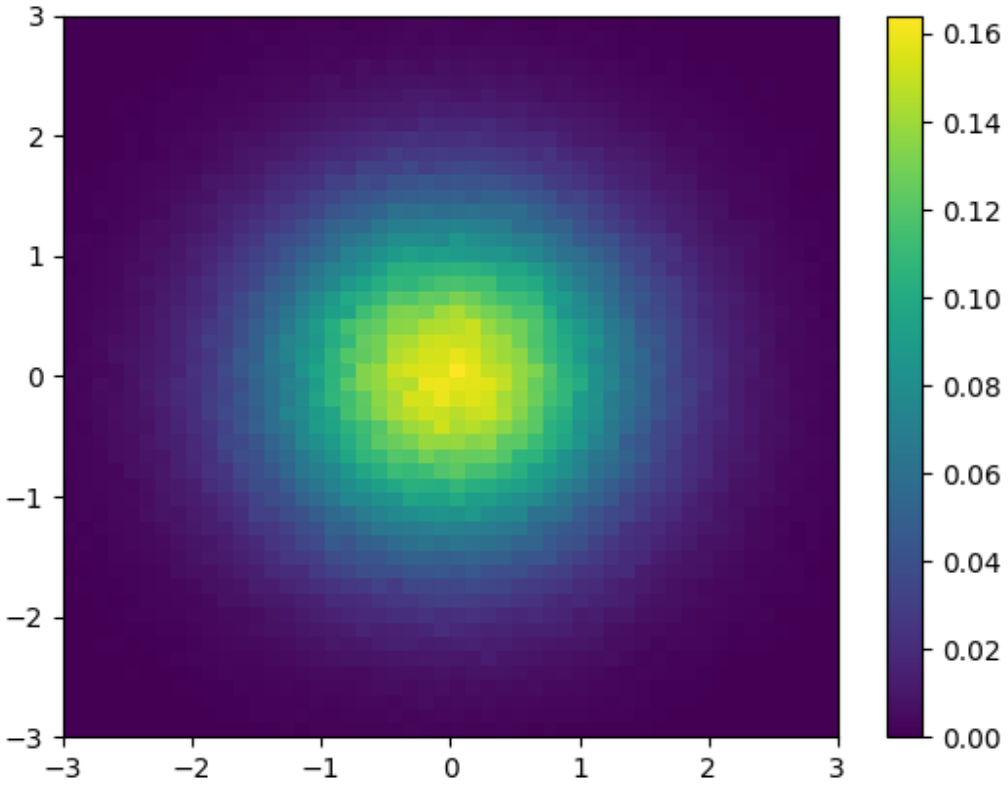
As an complete example, let's say you wanted to compute and plot the density, without using `.density()`:

```
import functools
import operator

import matplotlib.pyplot as plt
import numpy as np

import hist

# Make a 2D histogram
hist2d = hist.Hist(hist.axis.Regular(50, -3, 3), hist.axis.Regular(50, -3, 3))

# Fill with Gaussian random values
hist2d.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, hist2d.axes.widths)

# Compute the density
density = hist2d.values() / hist2d.sum() / areas
```

(continues on next page)

(continued from previous page)

```
# Make the plot
fig, ax = plt.subplots()
mesh = ax.pcolormesh(*hist2d.axes.edges.T, density.T)
fig.colorbar(mesh)
plt.savefig("simple_density.png")
```



1.3 Axes

In hist, a histogram is collection of Axis objects and a storage. Based on boost-histogram's Axis, hist support six types of axis, `Regular`, `Boolean`, `Variable`, `Integer`, `IntCategory` and `StrCategory` with additional names and labels.

1.3.1 Axis names

Names are pretty useful for some histogramming shortcuts, thus greatly facilitate HEP's studies. Note that the name is the identifier for an axis in a histogram and must be unique.

```
import hist
from hist import Hist
```

```
axis0 = hist.axis.Regular(10, -5, 5, overflow=False, underflow=False, name="A")
axis1 = hist.axis.Boolean(name="B")
axis2 = hist.axis.Variable(range(10), name="C")
axis3 = hist.axis.Integer(-5, 5, overflow=False, underflow=False, name="D")
axis4 = hist.axis.IntCategory(range(10), name="E")
axis5 = hist.axis.StrCategory(["T", "F"], name="F")
```

1.3.2 Histogram's Axis

Histogram is consisted with various axes, there are two ways to create a histogram, currently. You can either fill a histogram object with axes or add axes to a histogram object. You cannot add axes to an existing histogram. *Note that to distinguish these two method, the second way has different axis type names (abbr.).*

```
# fill the axes
h = Hist(axis0, axis1, axis2, axis3, axis4, axis5)
```

```
# add the axes using the shortcut method
h = (
    Hist.new.Reg(10, -5, 5, overflow=False, underflow=False, name="A")
    .Bool(name="B")
    .Var(range(10), name="C")
    .Int(-5, 5, overflow=False, underflow=False, name="D")
    .IntCat(range(10), name="E")
    .StrCat(["T", "F"], name="F")
    .Double()
)
```

Hist adds a new `flow=False` shortcut to axes that take `underflow` and `overflow`.

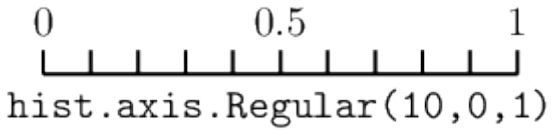
AxesTuple is a new feature since boost-histogram 0.8.0, which provides you free access to axis properties in a histogram.

```
assert h.axes[0].name == axis0.name
assert h.axes[1].label == axis1.name  # label will be returned as name if not provided
assert all(h.axes[2].widths == axis2.widths)
assert all(h.axes[3].edges == axis3.edges)
assert h.axes[4].metadata == axis4.metadata
assert all(h.axes[5].centers == axis5.centers)
```

1.3.3 Axis types

There are several axis types to choose from.

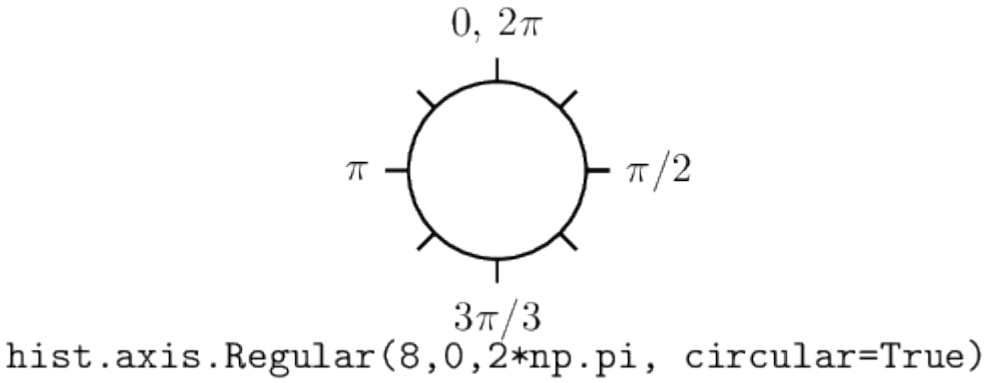
Regular axis



hist.axis.**Regular**(*bins*, *start*, *stop*, *name*, *label*, *, *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:



hist.axis.**Regular**(..., *circular=True*)

This wraps around, so that out-of-range values map back into the valid range in a circular fashion.

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 *Transform*. 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:

hist.axis.**Regular**(..., *transform=hist.axis.transform.sqrt*)

This is an axis with bins transformed by a sqrt.

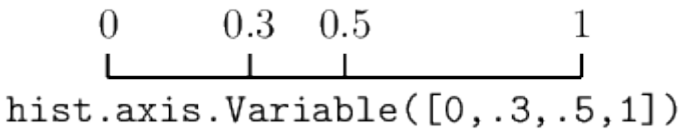
`hist.axis.Regular`(..., *transform=hist.axis.transform.log*)

Transformed by log.

`hist.axis.Regular`(..., *transform=hist.axis.transform.Power(v)*)

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

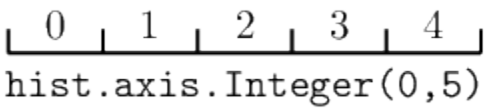
Variable axis



`hist.axis.Variable`([*edge1*, ...], *name*, *label*, *, *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.

Integer axis



`hist.axis.Integer`(*start*, *stop*, *name*, *label*, *, *metadata=''*, *underflow=True*, *overflow=True*, *circular=False*, *growth=False*)

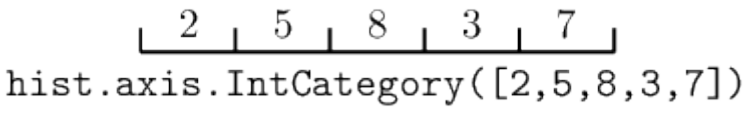
This could be mimicked with a regular axis, but is simpler and slightly faster. Bins are whole integers only, so there is no need to specify the number of bins.

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

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

Another could be for an IntEnum (Python 3 or backport) if the values are contiguous.

Category axis



`hist.axis.IntCategory`([*value1*, ...], *name*, *label*, *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):
```

(continues on next page)

(continued from previous page)

```
    a = 1
    b = 5


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

You can sort the Categorty axes via `.sort()` method:

```
h = Hist(
    axis.IntCategory([3, 1, 2], label="Number"),
    axis.StrCategory(["Teacher", "Police", "Artist"], label="Profession"),
)
# Sort Number axis increasing and Profession axis decreasing
h1 = h.sort("Number").sort("Profession", reverse=True)
```

`hist.axis.StrCategory`([*str1*, ...], *name*, *label*, *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.

1.3.4 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.

1.4 Storages

There are several storages to choose from. Based on boost-histogram's Storage, hist supports seven storage types, `Double`, `Unlimited`, `Int64`, `AutomicInt64`, `Weight`, `Mean`, and `WeightedMean`.

There are two methods to select a Storage in hist:

- **Method 1:** You can use boost-histogram's Storage in hist by passing the `storage=hist.storage.` argument when making a histogram.
- **Method 2:** Keeping the original features of boost-histogram's Storage, hist gives dynamic shortcuts of Storage Proxy. You can also add Storage types after adding the axes.

1.4.1 Simple Storages

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

Double

By default, hist 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.

```
# Method 1
h = Hist(hist.axis.Regular(10, -5, 5, name="x"), storage=hist.storage.Double())
h.fill([1.5, 2.5], weight=[0.5, 1.5])

print(h[1.5j])
print(h[2.5j])
```

```
0.5
1.5
```

```
# Method 2
h = (
    Hist.new.Reg(10, 0, 1, name="x")
    .Reg(10, 0, 1, name="y")
    .Double()
    .fill(x=[0.5, 0.5], y=[0.2, 0.6])
)

print(h[0.5j, 0.2j])
print(h[0.5j, 0.6j])
```

```
1.0
1.0
```

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.

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.

```
# Method 1
h = Hist(hist.axis.Regular(10, -5, 5, name="x"), storage=hist.storage.Int64())
h.fill([1.5, 2.5, 2.5, 2.5])

print(h[1.5j])
print(h[2.5j])
```

```
1
3
```

```
# Method 2
h = Hist.new.Reg(10, 0, 1, name="x").Int64().fill([0.5, 0.5])

print(h[0.5j])
```

```
2
```

AtomicInt64

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

1.4.2 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.

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.

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 can add a `sample=` term.

WeightedMean

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

1.5 Accumulators

1.5.1 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.

1.5.2 Types

There are several accumulators.

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 hist

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"{hist.accumulators.Sum().fill(values) = }")
```

```
sum(values) = 0.0 (simple)
math.fsum(values) = 2.0
np.sum(values) = 0.0 (pairwise)
hist.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"{hist.accumulators.Sum().fill(values) = }")
```

```
math.fsum(values) = 2.0
hist.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.

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 hist

values = [10]*10
```

(continues on next page)

(continued from previous page)

```
smooth = hist.accumulators.WeightedSum().fill(values)
print(f"{smooth = }")

values = [1]*9 + [91]
rough = hist.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 WeighedSum.

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 hist

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

values = [1]*9 + [91]
rough = hist.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.

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 hist

values = [1]*9 + [91]
wm = hist.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.

1.5.3 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.

1.6 Transform

Based on boost-histogram's Transform, hist 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:

- `hist.axis.transform.sqrt`: A square root transform;
- `hist.axis.transform.log`: A logarithmic transform;
- `hist.axis.transform.Pow(power)`: Raise to a specified power;
- `hist.axis.transform.Function`: Specify arbitrary conversion functions.

1.6.1 Transform Types

Here we show some basic usage of transforms in histogramming with pure Python. For more customized usage, you can refer the part in that of boost-histogram.

[1]:
```
import ctypes
import math

import numpy as np

import hist
from hist import Hist
```

[2]:
```
axis0 = hist.axis.Regular(10, 1, 4, name="A", transform=hist.axis.transform.sqrt)
axis1 = hist.axis.Regular(10, 1, 4, name="B", transform=hist.axis.transform.log)
axis2 = hist.axis.Regular(10, 1, 4, name="C", transform=hist.axis.transform.Pow(2))

ftype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
axis3 = hist.axis.Regular(
    10,
    1,
    4,
    name="D",
    transform=hist.axis.transform.Function(ftype(math.log), ftype(math.exp)),
)
```

(continues on next page)

(continued from previous page)

```
axis4 = hist.axis.Regular(
    10,
    1,
    4,
    name="E",
    transform=hist.axis.transform.Function(ftype(np.log), ftype(np.exp)),
)
```

[3]:
```
h = Hist(axis0, axis1, axis2, axis3, axis4)
```

1.6.2 Histogram's Transforms

Hist also provides transform shortcuts for histograms. hist's keeps the features of boost-histogram, and you can pass transform as a keyword argument when creating an axis. hist also allows you to use the `.new` shortcut directly when creating a Histogram for common transforms:

[4]:
```
h0 = Hist.new.Sqrt(10, 1, 4).Sqrt(10, 4, 9).Double()
h1 = Hist.new.Log(10, 1, 4).Log(10, 4, 9).Double()
h2 = Hist.new.Pow(10, 1, 4, power=3).Pow(10, 1, 4, power=5).Double()

h3 = (
    Hist.new.Func(10, 1, 4, forward=ftype(math.log), inverse=ftype(math.exp))
    .Func(10, 1, 4, forward=ftype(np.log), inverse=ftype(np.exp))
    .Double()
)
```

1.7 Indexing

Hist implements the UHI and UHI+ indexing protocols. You can read more about them on the Indexing and Indexing+ pages.

1.7.1 Hist specific details

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

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

Example:

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

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

(continues on next page)

(continued from previous page)

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

This feature is considered experimental. Removed bins are not added to the overflow bin currently.

1.8 Reprs in Jupyter

Hist has custom reprs when displaying in a Jupyter.

[1]:
```
import numpy as np

from hist import Hist
```

[2]:
```
Hist.new.Reg(50, 1, 2).Double().fill(np.random.normal(1.5, 0.3, 10_000))
```

[2]:
```
Hist(Regular(50, 1, 2, label='Axis 0'), storage=Double()) # Sum: 9101.0 (10000.0 with␣
↪flow)
```

[3]:
```
Hist.new.Reg(50, 0, 2 * 3, circular=True).Double().fill(
    np.random.normal(1.5, 0.7, 10_000)
)
```

[3]:
```
Hist(Regular(50, 0, 6, circular=True, label='Axis 0'), storage=Double()) # Sum: 10000.0
```

[4]:
```
Hist.new.Reg(50, 0, 2).Reg(50, 10, 20).Double().fill(
    np.random.normal(1, 0.5, 10_000), np.random.normal(15, 3, 10_000)
)
```

[4]:
```
Hist(
  Regular(50, 0, 2, label='Axis 0'),
  Regular(50, 10, 20, label='Axis 1'),
  storage=Double()) # Sum: 8675.0 (10000.0 with flow)
```

[5]:
```
Hist.new.Reg(50, 0, 2).Reg(50, 10, 20).Reg(2, 3, 4).Double()
```

[5]:
```
Hist(
  Regular(50, 0, 2, label='Axis 0'),
  Regular(50, 10, 20, label='Axis 1'),
  Regular(2, 3, 4, label='Axis 2'),
  storage=Double())
```

1.9 Plots

One of the most amazing feature of hist is it's powerful plotting family. Here you can see how to plot Hist.

[1]:
```
import hist
from hist import Hist
```

[2]:
```
h = Hist(
    hist.axis.Regular(50, -5, 5, name="S", label="s [units]", flow=False),
    hist.axis.Regular(50, -5, 5, name="W", label="w [units]", flow=False),
)
```

[3]:
```
import numpy as np

s_data = np.random.normal(size=100_000) + np.ones(100_000)
w_data = np.random.normal(size=100_000)

# normal fill
h.fill(s_data, w_data)
```

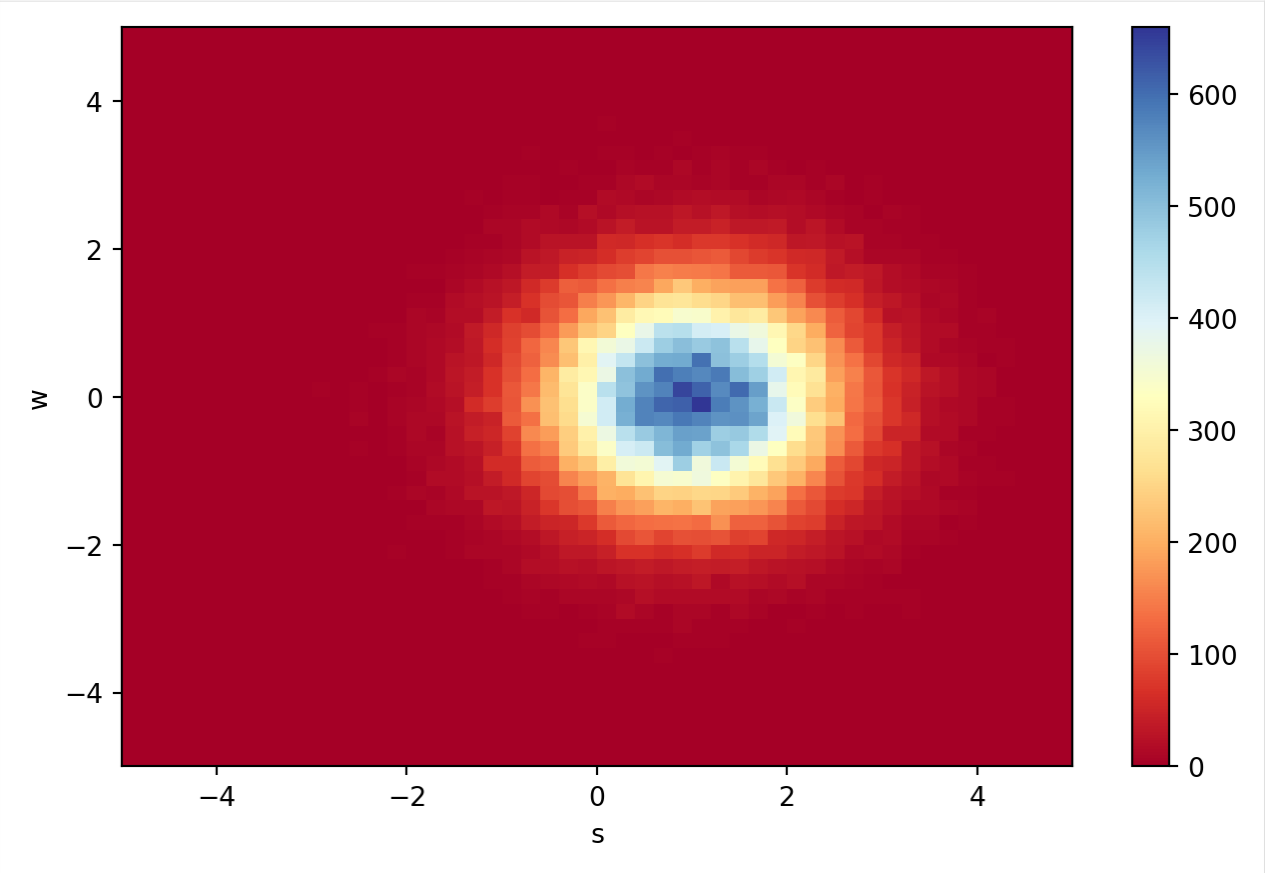
[3]:
```
Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 99997.0
```

1.9.1 Via Matplotlib

hist allows you to plot via Matplotlib like this:

[4]:
```
import matplotlib.pyplot as plt
```

[5]:
```
fig, ax = plt.subplots(figsize=(8, 5))
w, x, y = h.to_numpy()
mesh = ax.pcolormesh(x, y, w.T, cmap="RdYlBu")
ax.set_xlabel("s")
ax.set_ylabel("w")
fig.colorbar(mesh)
plt.show()
```

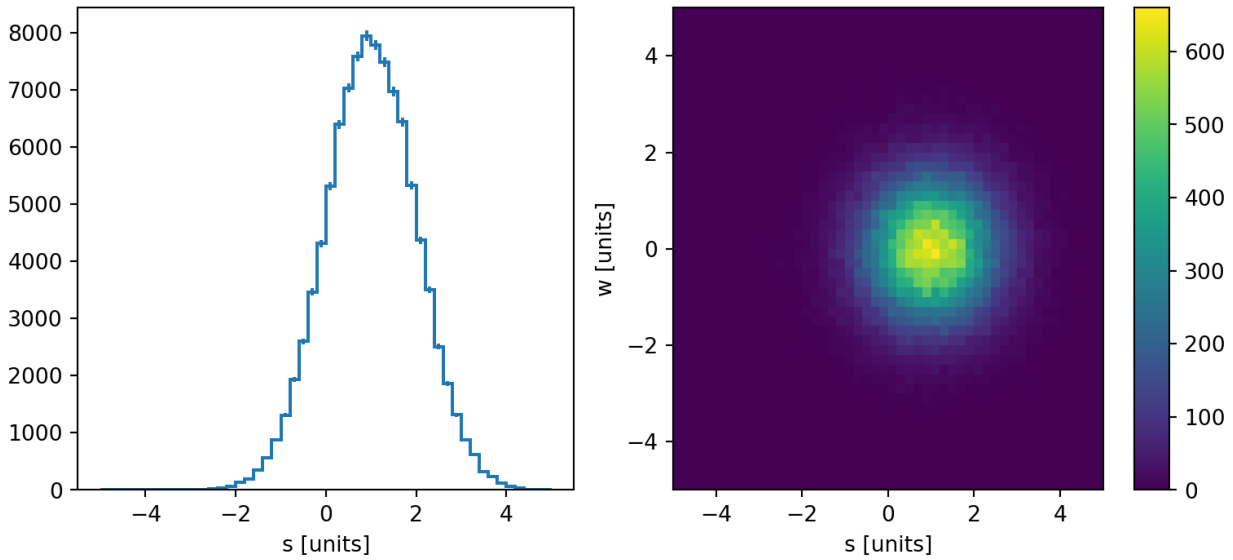


1.9.2 Via Mplhep

mplhep is an important visualization tools in Scikit-Hep ecosystem. hist has integrate with mplhep and you can also plot using it. If you want more info about mplhep please visit the official repo to see it.

[6]:
```
import mplhep

fig, axs = plt.subplots(1, 2, figsize=(9, 4))
mplhep.histplot(h.project("S"), ax=axs[0])

mplhep.hist2dplot(h, ax=axs[1])

plt.show()
```

```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 99997.0
```

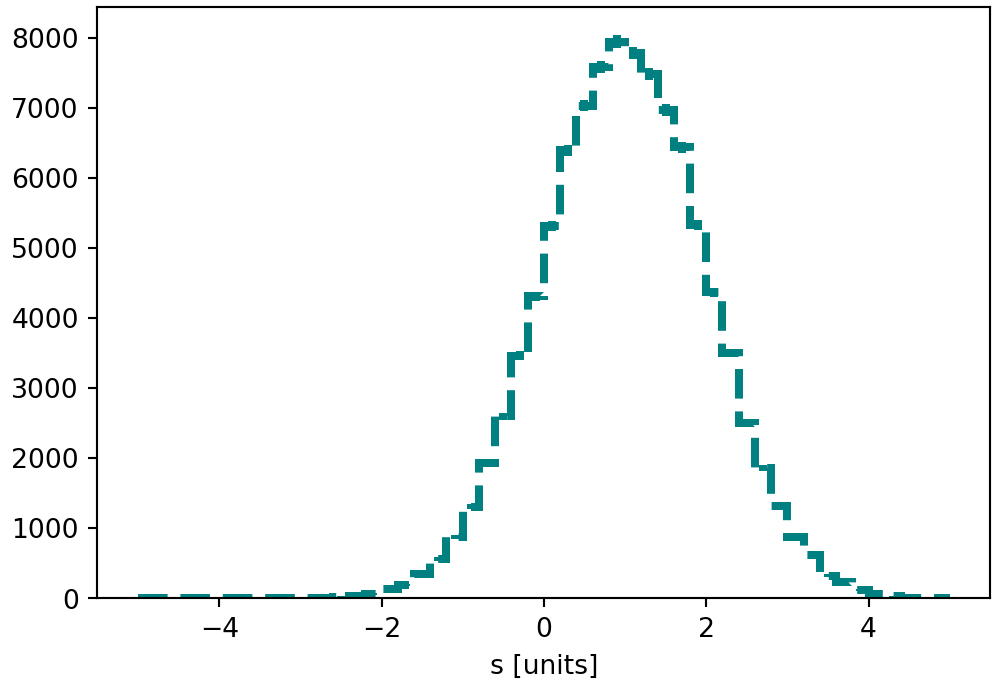


1.9.3 Via Plot

Hist has plotting methods for 1-D and 2-D histograms, `.plot1d()` and `.plot2d()` respectively. It also provides `.plot()` for plotting according to the its dimension. Moreover, to show the projection of each axis, you can use `.plot2d_full()`. If you have a Hist with higher dimension, you can use `.project()` to extract two dimensions to see it with our plotting suite.

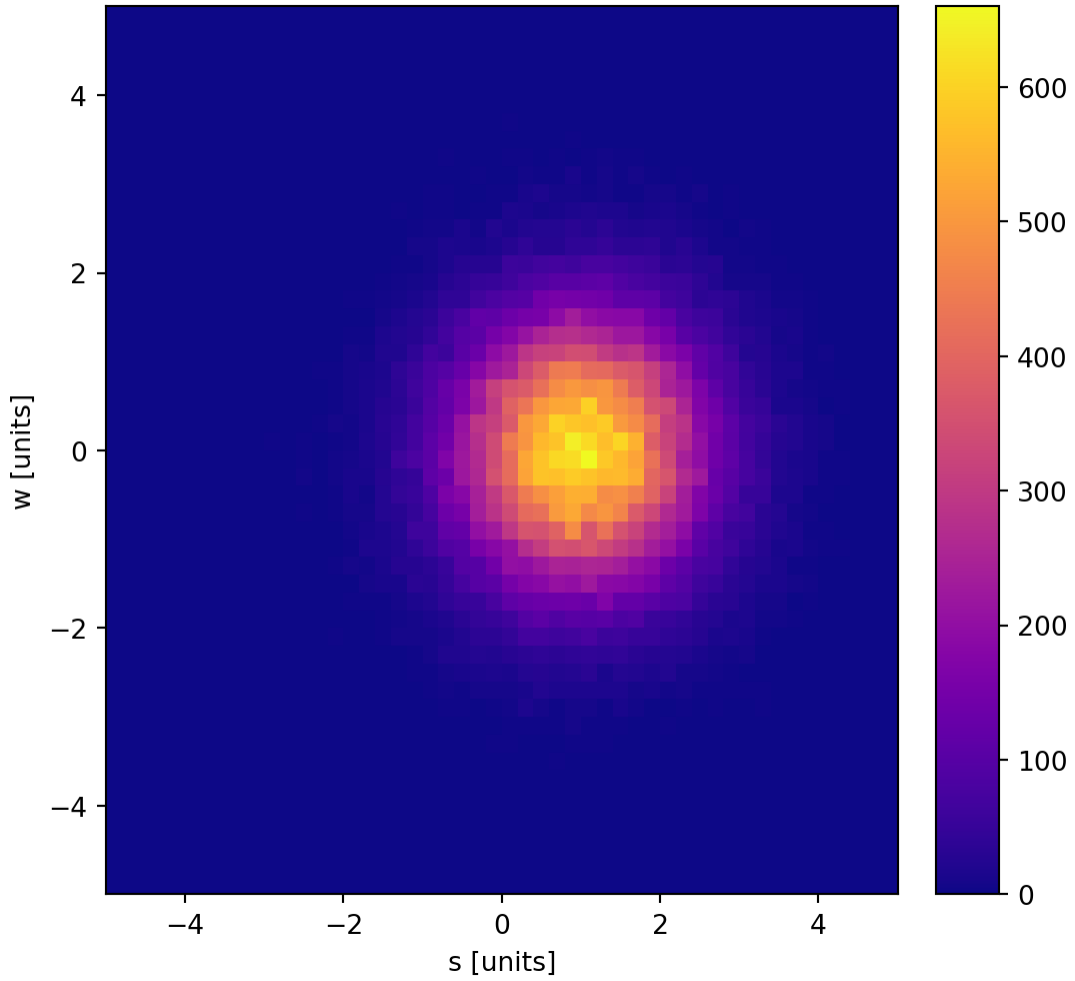
Our plotting methods are all based on Matplotlib, so you can pass Matplotlib's `ax` into it, and hist will draw on it. We will create it for you if you do not pass them in.

[7]:
```
# plot1d
fig, ax = plt.subplots(figsize=(6, 4))


h.project("S").plot1d(ax=ax, ls="--", color="teal", lw=3)
plt.show()
```



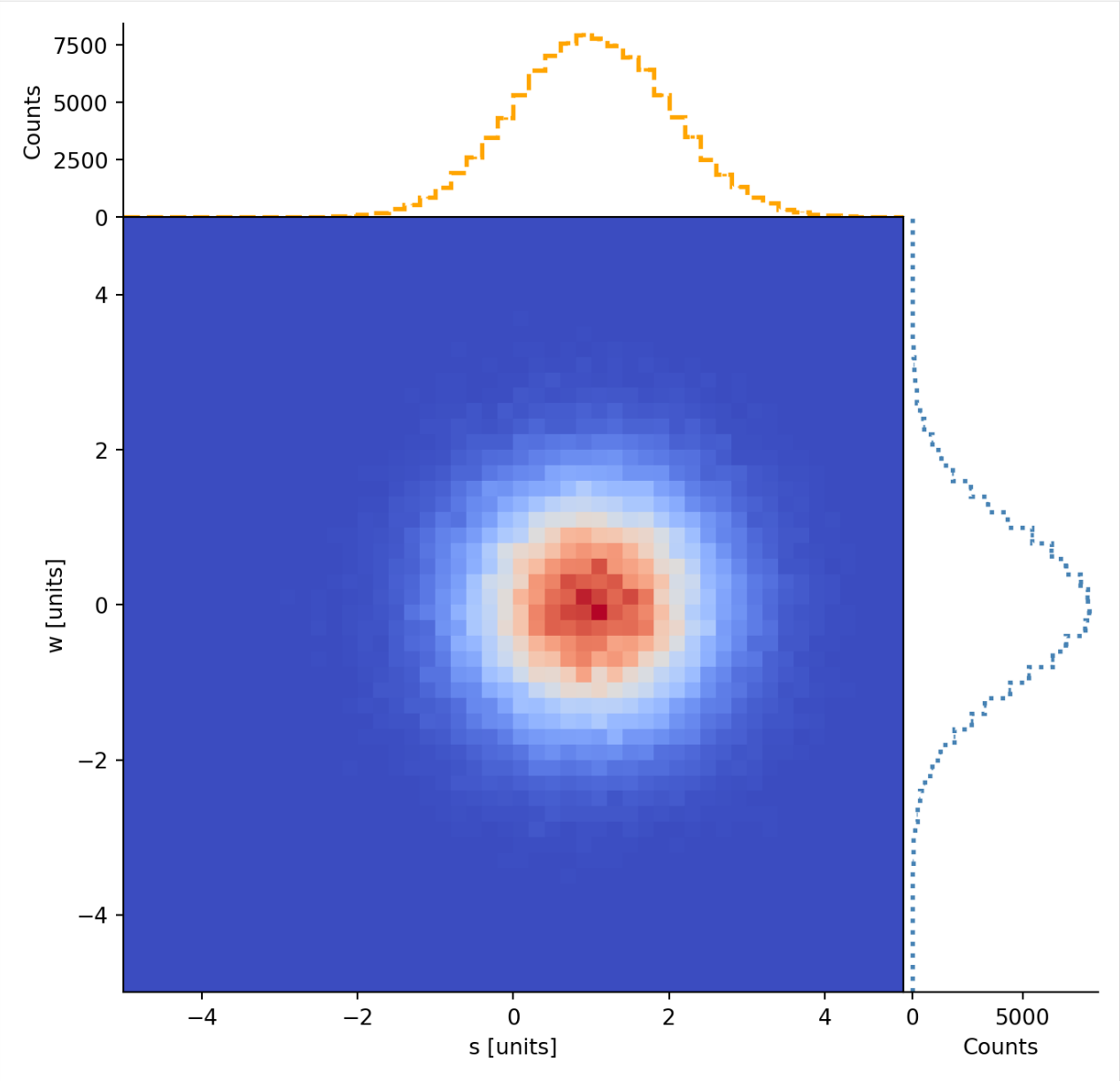
[8]:
```
# plot2d
fig, ax = plt.subplots(figsize=(6, 6))

h.plot2d(ax=ax, cmap="plasma")
plt.show()
```

```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 99997.0
```



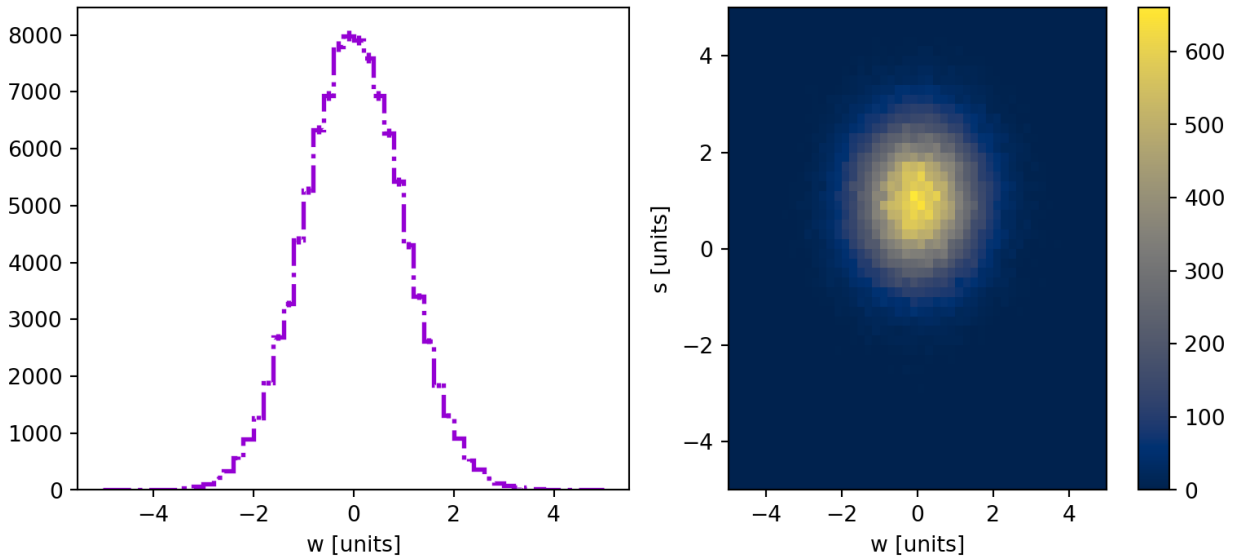
[9]:
```
# plot2d_full
plt.figure(figsize=(8, 8))

h.plot2d_full(
    main_cmap="coolwarm",
    top_ls="--",
    top_color="orange",
    top_lw=2,
    side_ls=":",
    side_lw=2,
    side_color="steelblue",
)

plt.show()
```

```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 99997.0
```



[10]:
```
# auto-plot
fig, axs = plt.subplots(1, 2, figsize=(9, 4), gridspec_kw={"width_ratios": [5, 4]})


h.project("W").plot(ax=axs[0], color="darkviolet", lw=2, ls="-.")
h.project("W", "S").plot(ax=axs[1], cmap="cividis")

plt.show()
```

```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  storage=Double()) # Sum: 99997.0
```



1.9.4 Via Plot Pull

Pull plots are commonly used in HEP studies, and we provide a method for them with `.plot_pull()`, which accepts a `Callable` object, like the below `pdf` function, which is then fit to the histogram and the fit and pulls are shown on the plot. As Normal distributions are the generally desired function to fit the histogram data, the `str` aliases `"normal"`, `"gauss"`, and `"gaus"` are supported as well.

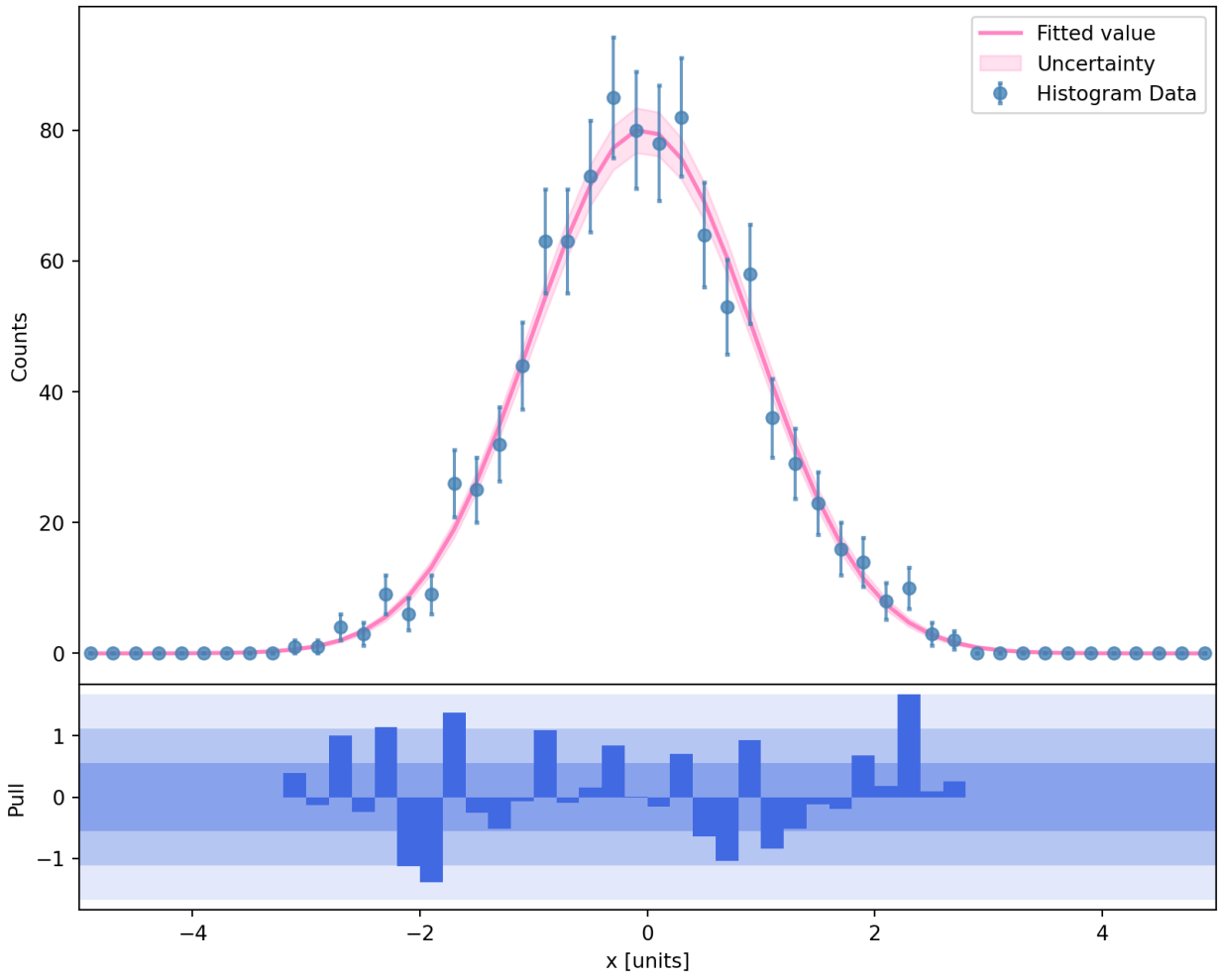
[11]:
```
def pdf(x, a=1 / np.sqrt(2 * np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-((x - x0) ** 2) / (2 * sigma**2)) + offset
```

[12]:
```
np.random.seed(0)

hist_1 = hist.Hist(
    hist.axis.Regular(
        50, -5, 5, name="X", label="x [units]", underflow=False, overflow=False
    )
).fill(np.random.normal(size=1000))

fig = plt.figure(figsize=(10, 8))
main_ax_artists, sublot_ax_arists = hist_1.plot_pull(
    "normal",
    eb_ecolor="steelblue",
    eb_mfc="steelblue",
    eb_mec="steelblue",
    eb_fmt="o",
    eb_ms=6,
    eb_capsize=1,
    eb_capthick=2,
    eb_alpha=0.8,
    fp_c="hotpink",
    fp_ls="-",
    fp_lw=2,
```

(continues on next page)

(continued from previous page)

```
    fp_alpha=0.8,
    bar_fc="royalblue",
    pp_num=3,
    pp_fc="royalblue",
    pp_alpha=0.618,
    pp_ec=None,
    ub_alpha=0.2,
)
```



1.9.5 Via Plot Ratio

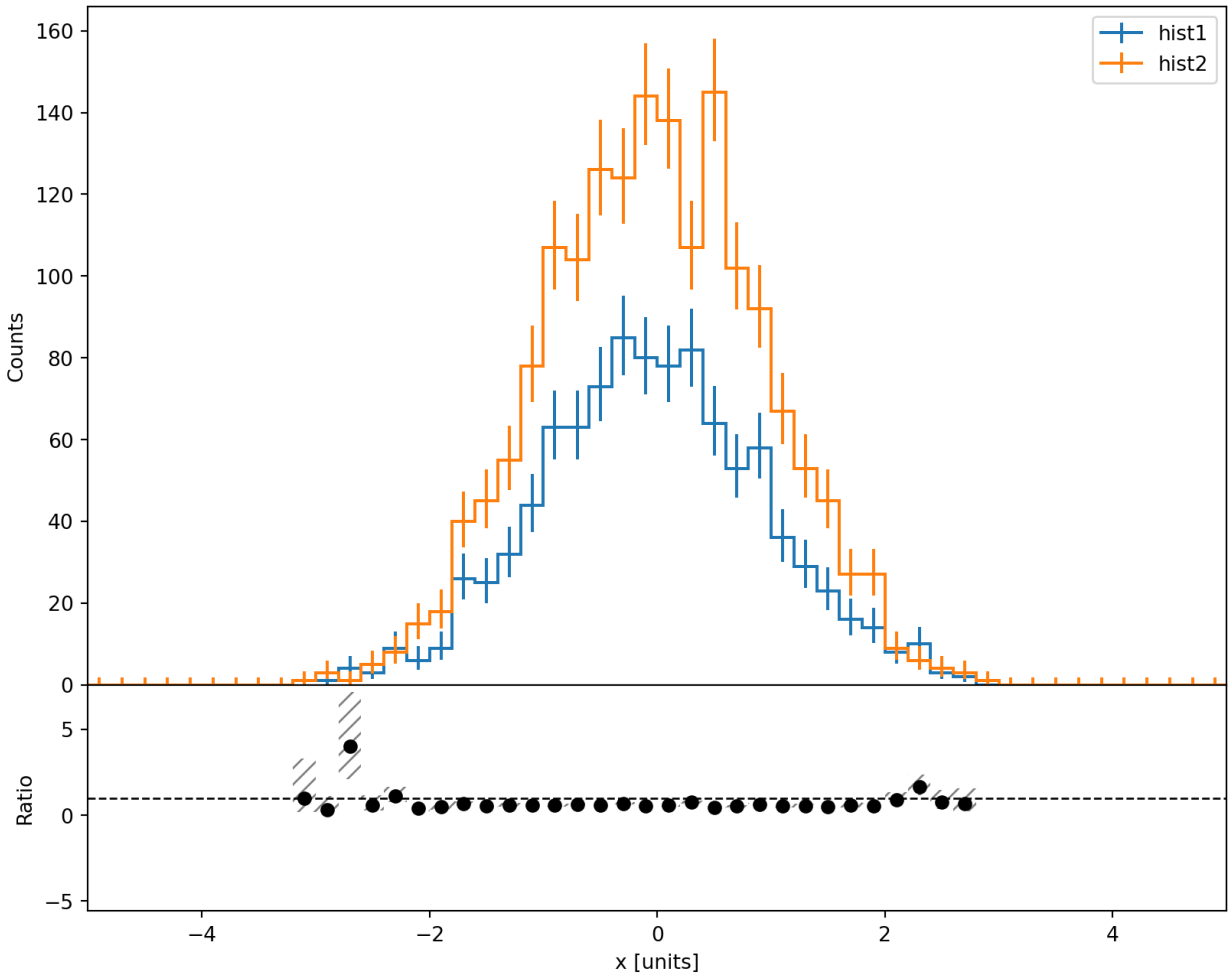
You can also make an arbitrary ratio plot using the `.plot_ratio` API:

[13]:
```
hist_2 = hist.Hist(
    hist.axis.Regular(
        50, -5, 5, name="X", label="x [units]", underflow=False, overflow=False
    )
).fill(np.random.normal(size=1700))
```

(continues on next page)

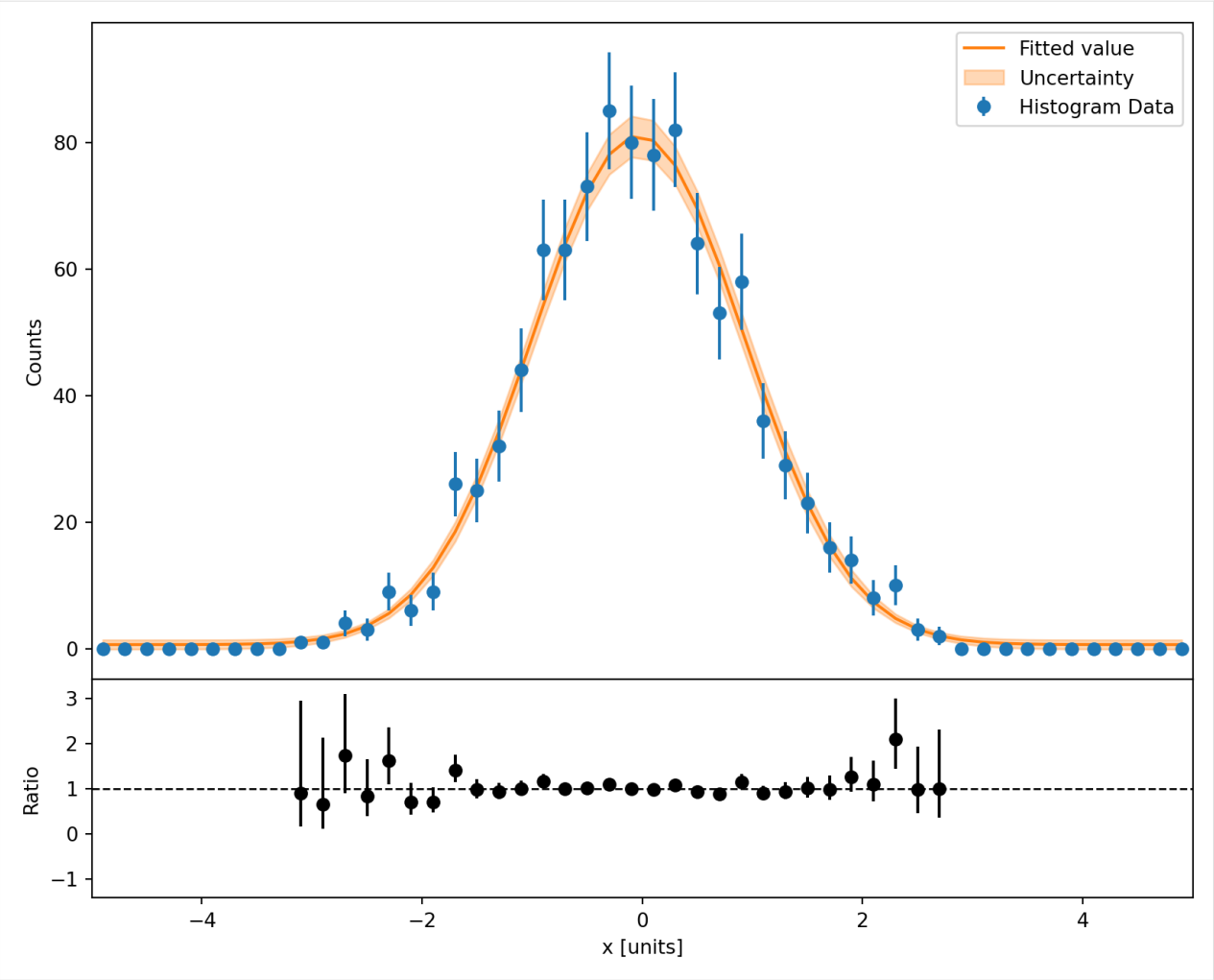
(continued from previous page)

```
fig = plt.figure(figsize=(10, 8))
main_ax_artists, sublot_ax_arists = hist_1.plot_ratio(
    hist_2,
    rp_ylabel=r"Ratio",
    rp_num_label="hist1",
    rp_denom_label="hist2",
    rp_uncert_draw_type="bar",  # line or bar
)
```



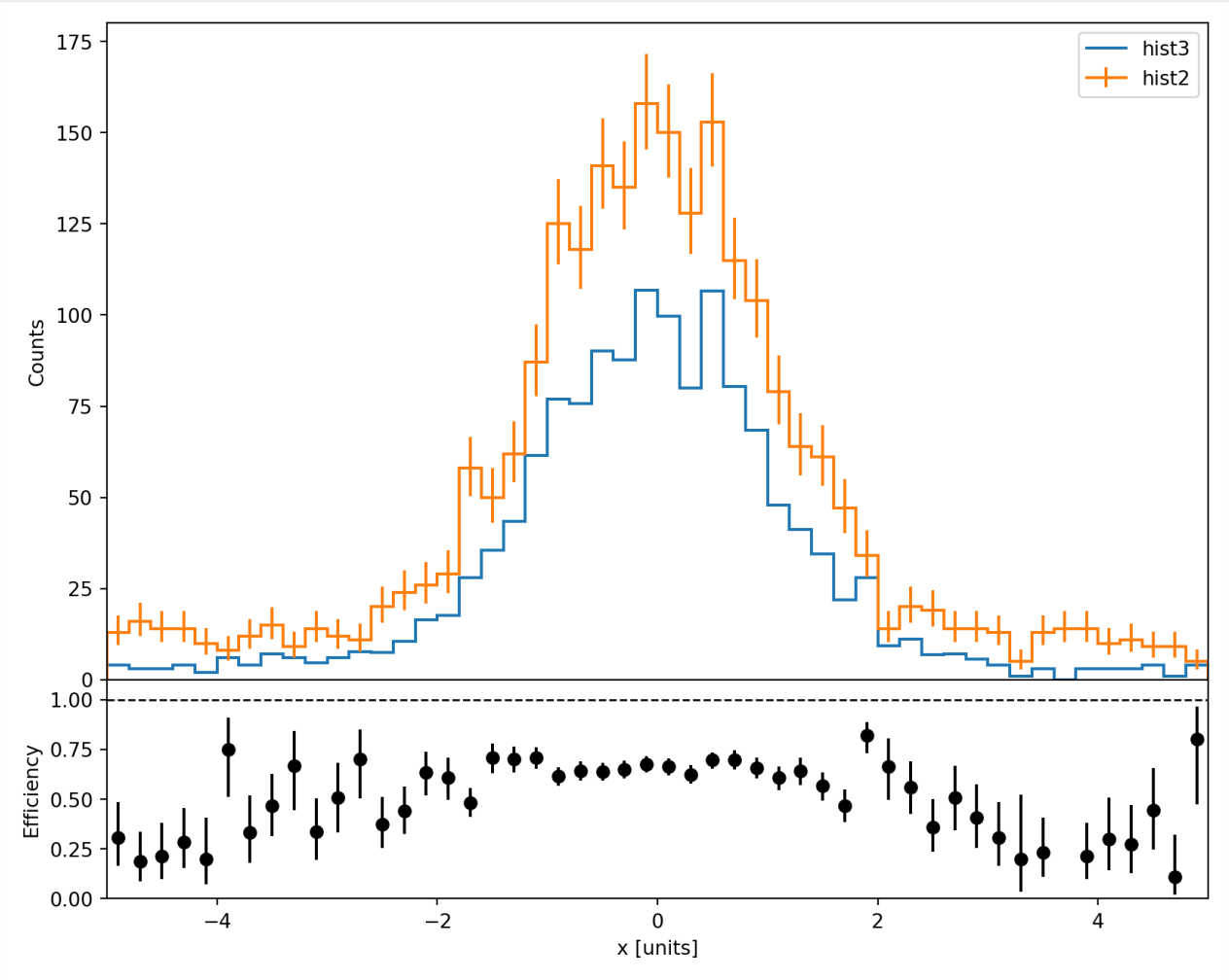
Ratios between the histogram and a callable, or `str` alias, are supported as well

[14]:
```
fig = plt.figure(figsize=(10, 8))
main_ax_artists, sublot_ax_arists = hist_1.plot_ratio(pdf)
```



Using the `.plot_ratio` API you can also make efficiency plots (where the numerator is a strict subset of the denominator)

[15]:
```
hist_3 = hist_2.copy() * 0.7
hist_2.fill(np.random.uniform(-5, 5, 600))
hist_3.fill(np.random.uniform(-5, 5, 200))

fig = plt.figure(figsize=(10, 8))
main_ax_artists, sublot_ax_arists = hist_3.plot_ratio(
    hist_2,
    rp_num_label="hist3",
    rp_denom_label="hist2",
    rp_uncert_draw_type="line",
    rp_uncertainty_type="efficiency",
)
```



1.10 Analyses examples

1.10.1 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 hist, you can make a single histogram, and use an axis for each:

```
value_ax = hist.axis.Regular(100, -5, 5)
bool_ax = hist.axis.Integer(0, 2, underflow=False, overflow=False)
run_number_ax = hist.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.

1.11 NumPy compatibility

1.11.1 Histogram conversion

Accessing the storage array

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

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).

1.11.2 NumPy adaptors

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

1D histogram example

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

```
import numpy as np
import hist

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, hist has adaptors for the three histogram functions in NumPy:

```
%%timeit
bins, edges = hist.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 hist API, and is still a nice performance boost over NumPy. You can even use the NumPy syntax if you want a hist object later:

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

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 hist.

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
hist.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, hist is more than an order of magnitude faster than NumPy for regular spaced binning. Although optimizations may be added to hist for common axes combinations later, 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.

1.12 Subclassing (advanced)

Subclassing hist components is supported, but requires a little extra care to ensure the subclasses do not return unwrapped hist 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. Hist uses `hist`, Boost-histogram uses `boost_histogram`. 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 hist
import my_package


class Histogram(hist.Hist, family=my_package):
    ...


class Regular(hist.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.

1.13 Histogram

hist provides two types of histograms, in which Hist is the general class, NamedHist is a forced-name class. hist supports the whole workflow for a histogram's lifecycle, including some plotting tools and shortcuts which are pretty useful for HEP studies. Here, you can see how to serialize/deserialize (will be achieved), construct, use, and visualize histograms.

1.13.1 Hist

Hist is the general class in the hist package based on boost-histogram's Histogram. Here is how to serialize/deserialize (will be achieved), construct, use, and visualize histograms via Hist.

Initialize Hist

You need to initialized Hist first before you use it. Two ways are provided: you can just fill the axes into the Hist instance and create it; you can also add axes in Hist object via hist proxy.

When initializing you don't have to use named-axes, axes without names are allowed. Using named-axes is recommended, because you will get more shortcuts to make the best of hist (there is also a classed called NamedHist which forces names be used most places). Duplicated non-empty names are not allowed in the Hist as name is the unique identifier for a Hist object.

[1]:
```
import hist
from hist import Hist
```

Standard method:

[2]:
```
# fill the axes
h = Hist(
    hist.axis.Regular(
        50, -5, 5, name="S", label="s [units]", underflow=False, overflow=False
    ),
    hist.axis.Regular(
        50, -5, 5, name="W", label="w [units]", underflow=False, overflow=False
    ),
)
```

Shortcut method:

One benefit of the shortcut method is that you can work entirely from Hist, so `from hist import Hist` can be used.

[3]:
```
# add the axes, finalize with storage
h = (
    Hist.new.Reg(50, -5, 5, name="S", label="s [units]", flow=False)
    .Reg(50, -5, 5, name="W", label="w [units]", flow=False)
    .Double()
)
```

Manipulate Hist

Fill Hist

After initializing the Hist, the most likely thing you want to do is to fill it. The normal method to fill the histogram is just to pass the data to `.fill()`, and the data will be filled in the index order. If you have axes all with names in your Hist, you will have another option – filling by names in the order of names given.

[4]:
```
import numpy as np

s_data = np.random.normal(size=50_000)
w_data = np.random.normal(size=50_000)

# normal fill
h.fill(s_data, w_data)

# Clear the data since we want to fill again with the same data
h.reset()

# fill by names
h.fill(W=w_data, S=s_data)
```

[4]:
```
Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 50000.0
```

Access Bins

hist allows you to access the bins of your Hist by various ways. Besides the normal access by index, you can use locations (supported by boost-histogram), complex numbers, and the dictionary to access the bins.

[5]:
```
# Access by bin number
h[25, 25]
```

[5]:
```
304.0
```

[6]:
```
# Access by data coordinate
# Identical to: h[hist.loc(0), hist.loc(0)]
h[0j, 0j]
```

[6]:
```
304.0
```

[7]:
```
# Identical to: h[hist.loc(-1) + 5, hist.loc(-4) + 20]
h[-1j + 5, -4j + 20]
```

[7]:
```
304.0
```

If you are accessing multiple bins, you can use complex numbers to rebin.

[8]:
```
# Identical to: h.project("S")[20 : 30 : hist.rebin(2)]
h.project("S")[20:30:2j]
```

[8]:
```
Hist(Regular(5, -1, 1, underflow=False, overflow=False, name='S', label='s [units]'),␣
↪storage=Double()) # Sum: 34098.0
```

Dictionary is allowed when accessing bins. If you have axes all with names in your Hist, you can also access them according to the axes' names.

[9]:
```
s = Hist(
    hist.axis.Regular(50, -5, 5, name="Norm", label="normal distribution"),
    hist.axis.Regular(50, 0, 1, name="Unif", label="uniform distribution"),
    hist.axis.StrCategory(["hi", "hello"], name="Greet"),
    hist.axis.Boolean(name="Yes"),
    hist.axis.Integer(0, 1000, name="Int"),
)
```

[10]:
```
s.fill(
    Norm=np.random.normal(size=1000),
    Unif=np.random.uniform(size=1000),
    Greet=["hi"] * 800 + ["hello"] * 200,
    Yes=[True] * 600 + [False] * 400,
    Int=np.ones(1000, dtype=int),
)
```

[10]:
```
Hist(
  Regular(50, -5, 5, name='Norm', label='normal distribution'),
  Regular(50, 0, 1, name='Unif', label='uniform distribution'),
  StrCategory(['hi', 'hello'], name='Greet'),
  Boolean(name='Yes'),
  Integer(0, 1000, name='Int'),
  storage=Double()) # Sum: 1000.0
```

[11]:
```
s[0j, -0j + 2, "hi", True, 1]
```

[11]:
```
1.0
```

[12]:
```
s[{0: 0j, 3: True, 4: 1, 1: -0j + 2, 2: "hi"}] += 10

s[{"Greet": "hi", "Unif": -0j + 2, "Yes": True, "Int": 1, "Norm": 0j}]
```

[12]:
```
11.0
```

Get Density

If you want to get the density of an existing histogram, `.density()` is capable to do it and will return you the density array without overflow and underflow bins. (*This may return a "smart" object in the future; for now it's a simple NumPy array.*)

[13]:
```
h[25:30, 25:30].density()
```

[13]:
```
array([[1.31533403, 1.23312565, 1.14226376, 1.08601592, 0.80044998],
       [1.33264105, 1.3283143 , 1.3283143 , 1.00380755, 0.83073728],
       [1.37590862, 1.14659052, 1.09899619, 0.8091035 , 0.76583593],
       [1.09899619, 0.89996539, 0.9086189 , 0.92592593, 0.70093458],
       [0.83939079, 0.98217376, 0.75718242, 0.6793008 , 0.61007269]])
```

Get Project

Hist allows you to get the projection of an N-D Histogram:

[14]:
```
s_2d = s.project("Norm", "Unif")
s_2d
```

[14]:
```
Hist(
  Regular(50, -5, 5, name='Norm', label='normal distribution'),
  Regular(50, 0, 1, name='Unif', label='uniform distribution'),
  storage=Double()) # Sum: 1010.0
```

Get Profile

To compute the (N-1)-D profile from an existing histogram, you can:

[15]:
```
xy = np.array(
    [
        [-2, 1.5],
        [-2, -3.5],
        [-2, 1.5],  # x = -2
        [0.0, -2.0],
        [0.0, -2.0],
        [0.0, 0.0],
        [0.0, 2.0],
        [0.0, 4.0],  # x = 0
        [2, 1.5],  # x = +2
    ]
)
h_xy = hist.Hist(
    hist.axis.Regular(5, -5, 5, name="x"), hist.axis.Regular(5, -5, 5, name="y")
).fill(*xy.T)

# Profile out the y-axis
hp = h_xy.profile("y")
hp.values()[1:-1]
# hp.variances()[1:-1]
```

[15]:
```
array([-1.48029737e-16,  4.00000000e-01,  2.00000000e+00])
```

Plot Hist

One of the most amazing feature of hist is it's powerful plotting family. Here is a brief demonstration of how to plot Hist. You can get more information in the section of *Plots*.

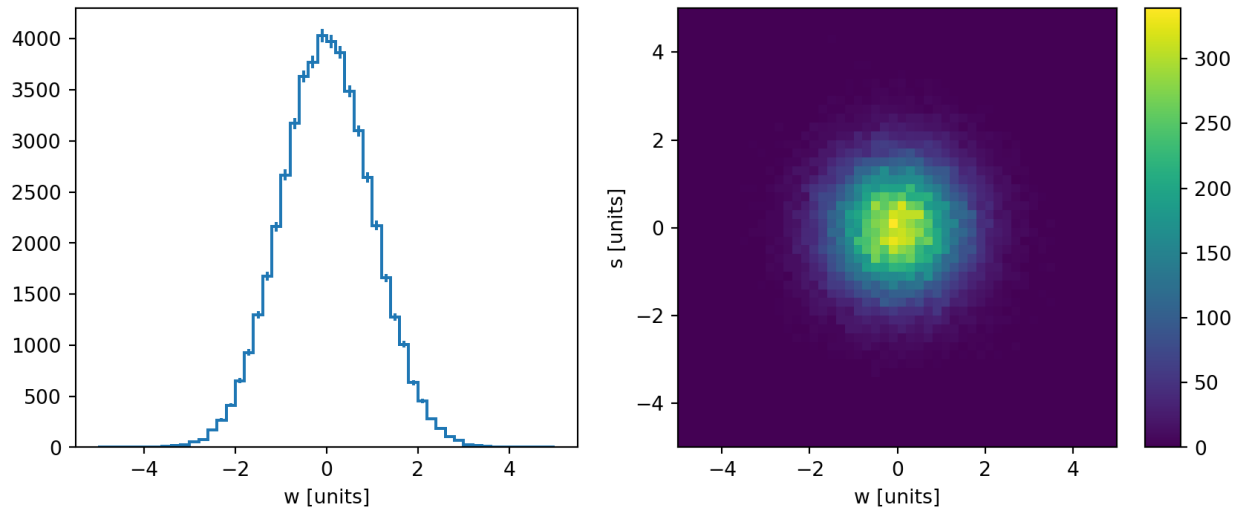
[16]:
```
import matplotlib.pyplot as plt

# auto-plot
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

h.project("W").plot(ax=axs[0])
```

(continues on next page)

(continued from previous page)

```
h.project("W", "S").plot(ax=axs[1])
plt.show()
```

```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  storage=Double()) # Sum: 50000.0
```

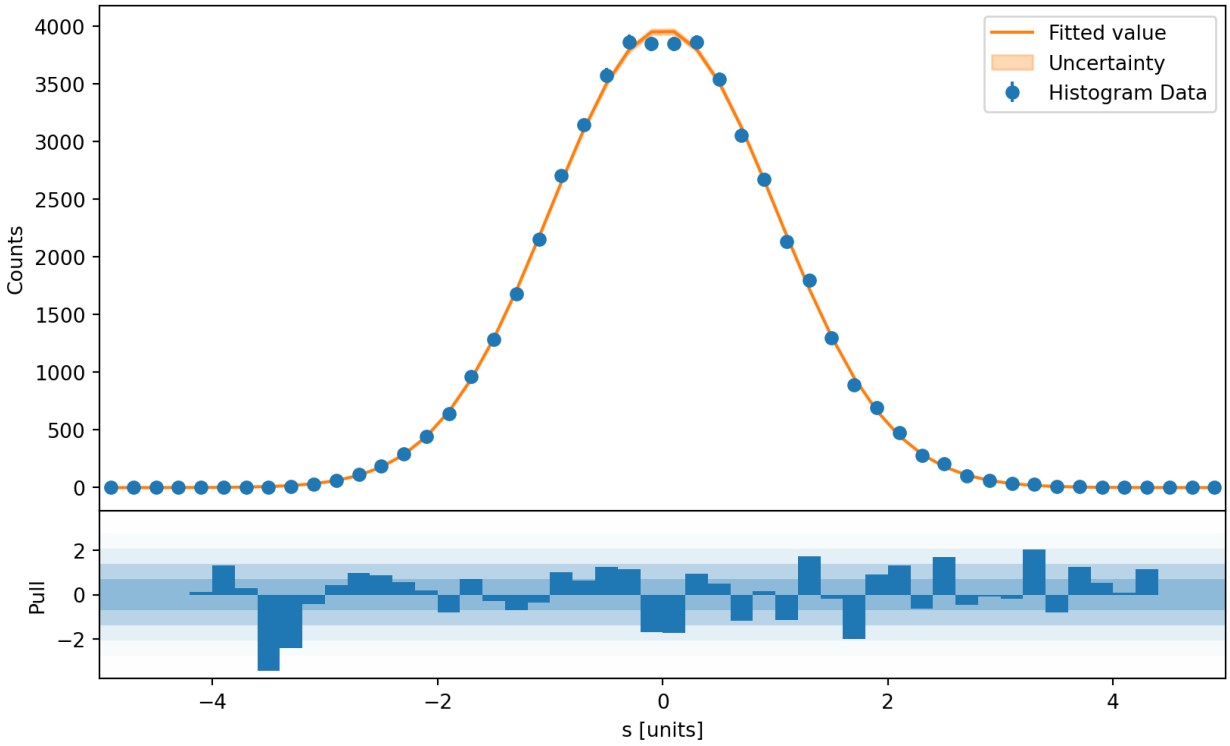


This is an example of a pull plot:

[17]:
```
from uncertainties import unumpy as unp


def pdf(x, a=1 / np.sqrt(2 * np.pi), x0=0, sigma=1, offset=0):
    exp = unp.exp if a.dtype == np.dtype("O") else np.exp
    return a * exp(-((x - x0) ** 2) / (2 * sigma**2)) + offset
```

(The uncertainty is non-significant as we filled a great quantities of observation points above.)

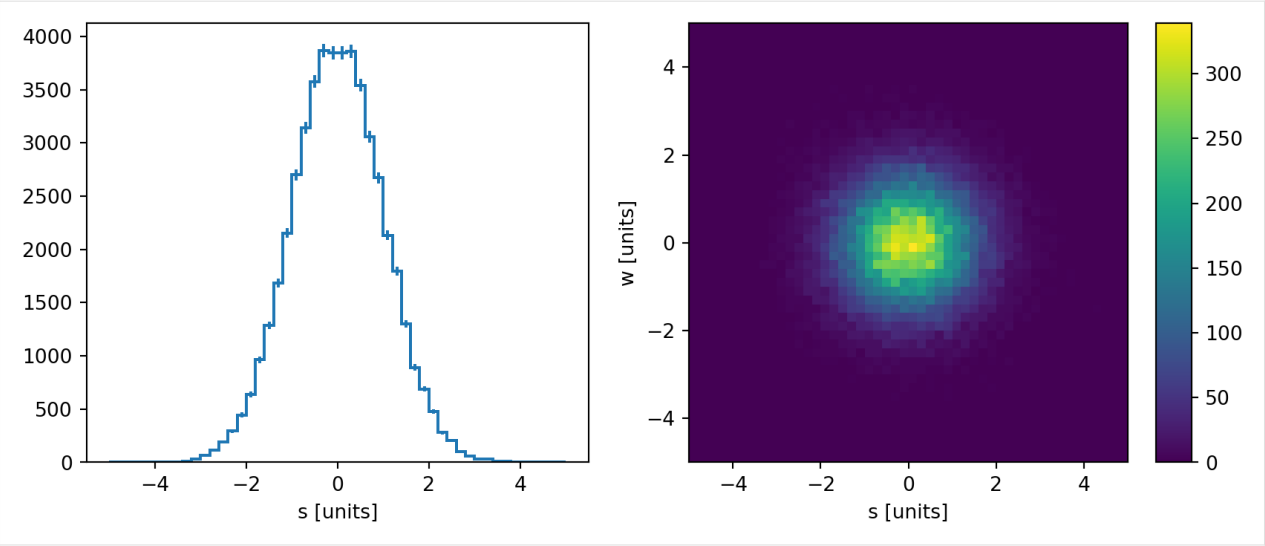
[18]:
```
plt.figure(figsize=(10, 6))

h.project("S").plot_pull(pdf)

plt.show()
```



You can also pass Hist objects directly to mplhep (which is what is used for the backend of Hist anyway):

[19]:
```
import mplhep

# auto-plot
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

mplhep.histplot(h.project("S"), ax=axs[0])
mplhep.hist2dplot(h, ax=axs[1])

plt.show()
```

```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 50000.0
```



1.13.2 NamedHist

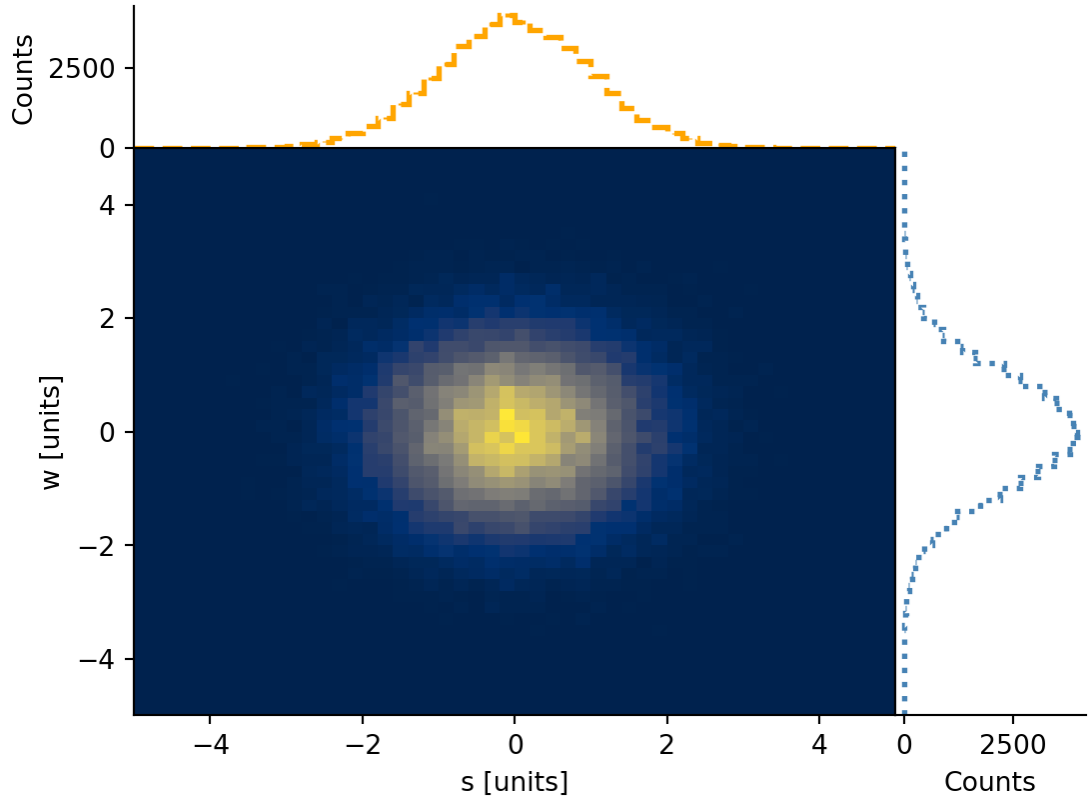
If you want to force names always be used, you can use NamedHist. This reduces functionality but can reduce mistaking one axes for another.

[20]:
```
h = hist.NamedHist(
    hist.axis.Regular(
        50, -5, 5, name="S", label="s [units]", underflow=False, overflow=False
    ),
    hist.axis.Regular(
        50, -5, 5, name="W", label="w [units]", underflow=False, overflow=False
    ),
)
```

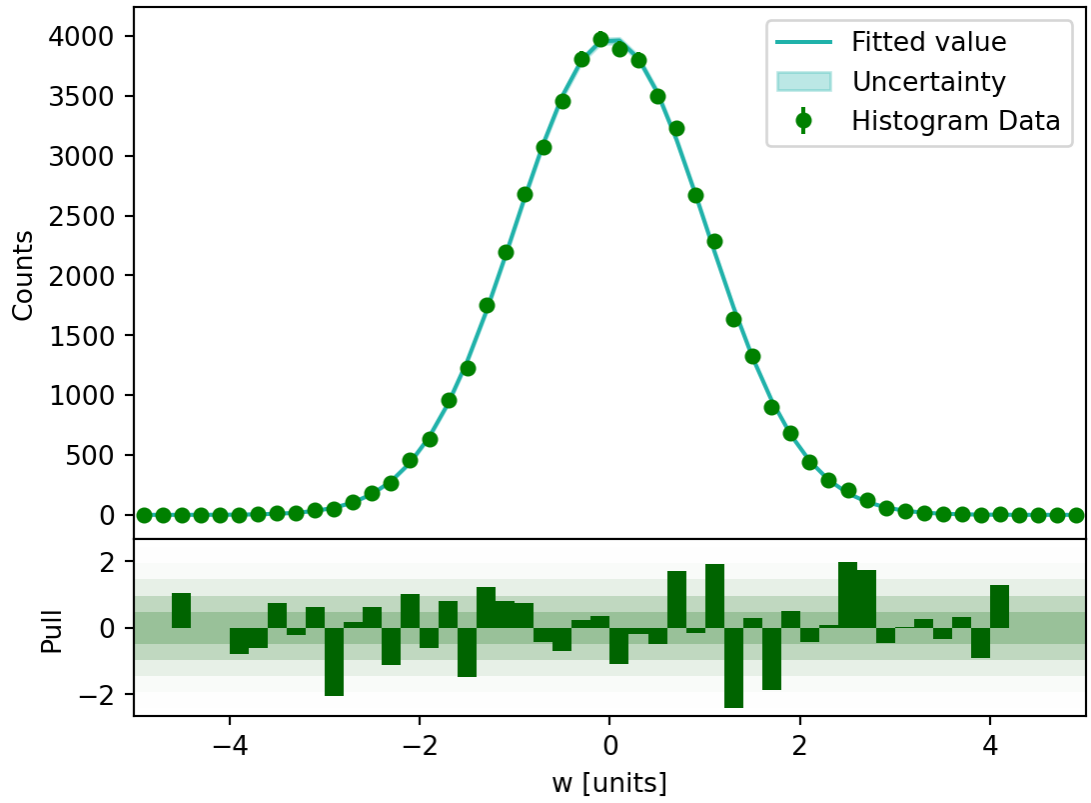
[21]:
```
# should all use names
s_data = np.random.normal(size=50_000)
w_data = np.random.normal(size=50_000)

h.fill(W=w_data, S=s_data)

assert h[25, 25] == h[0j, 1j - 5] == h[{"W": 25, "S": 0j}]
assert h[:, 0:50:5j].project("S")
```

[22]:
```
# plot2d_full
h.plot2d_full(
    main_cmap="cividis",
    top_ls="--",
    top_color="orange",
    top_lw=2,
    side_ls=":",
    side_lw=2,
    side_color="steelblue",
)
plt.show()
```

```
Warning:  you don't have flow bins stored in NamedHist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 50000.0
```



[23]:
```
# plot pull
h.project("W").plot_pull(
    pdf,
    eb_ecolor="green",
    eb_mfc="green",
    eb_mec="green",
    eb_fmt="o",
    eb_ms=5,
    fp_color="lightseagreen",
    pp_color="darkgreen",
    pp_alpha=0.4,
    pp_ec=None,
    bar_color="darkgreen",
)
plt.show()
```



1.13.3 hist.dask

If you want to fill your histograms using delayed arrays provided by dask start by importing the hist.dask sub-package, usually calling it `dah`. Within this sub-package dask versions of Hist and NamedHist are available. All methods of Hist and NamedHist instantiation discussed above are supported in their dask forms. This method of using hist can be best used when operating on large datasets and distributed clusters.

An important note: as with all dask collections the in-memory and finalized form of the histogram is only rendered when you call `.compute()` or `dask.compute()` on the dask collection! Until that point you are manipulating a *task graph* that represents the process of filling and creating that histogram.

[24]:
```
import dask.array as da

import hist.dask as dah
```

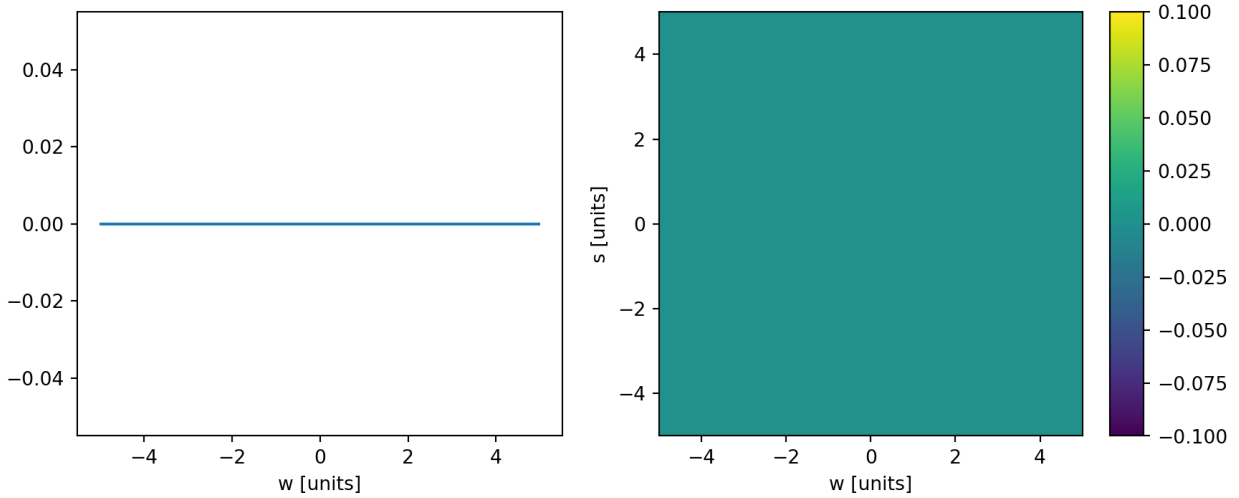
Hist

Below we'll use a dask array to fill a `hist.dask.Hist` lazily, as a proxy for filling it on a cluster, and then plot the resulting histogram!

[25]:
```
# add the axes, finalize with storage
h = (
    dah.Hist.new.Reg(50, -5, 5, name="S", label="s [units]", flow=False)
    .Reg(50, -5, 5, name="W", label="w [units]", flow=False)
    .Double()
)

s_data = da.random.standard_normal(size=(50_000,), chunks=(1000,))
w_data = da.random.standard_normal(size=(50_000,), chunks=(1000,))

# delayed fill
h.fill(W=w_data, S=s_data)

import matplotlib.pyplot as plt

# auto-plot
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

h.project("W").plot(ax=axs[0])
h.project("W", "S").plot(ax=axs[1])
plt.show()

h.visualize()  # from here we can see that only the task graph is created and there is␣
↪no filled histogram!
```

```
/home/docs/checkouts/readthedocs.org/user_builds/hist/envs/stable/lib/python3.10/site-
↪packages/mplhep/utils.py:197: RuntimeWarning: All sumw are zero!  Cannot compute␣
↪meaningful error bars
  return np.abs(method_fcn(self.values, variances) - self.values)
```

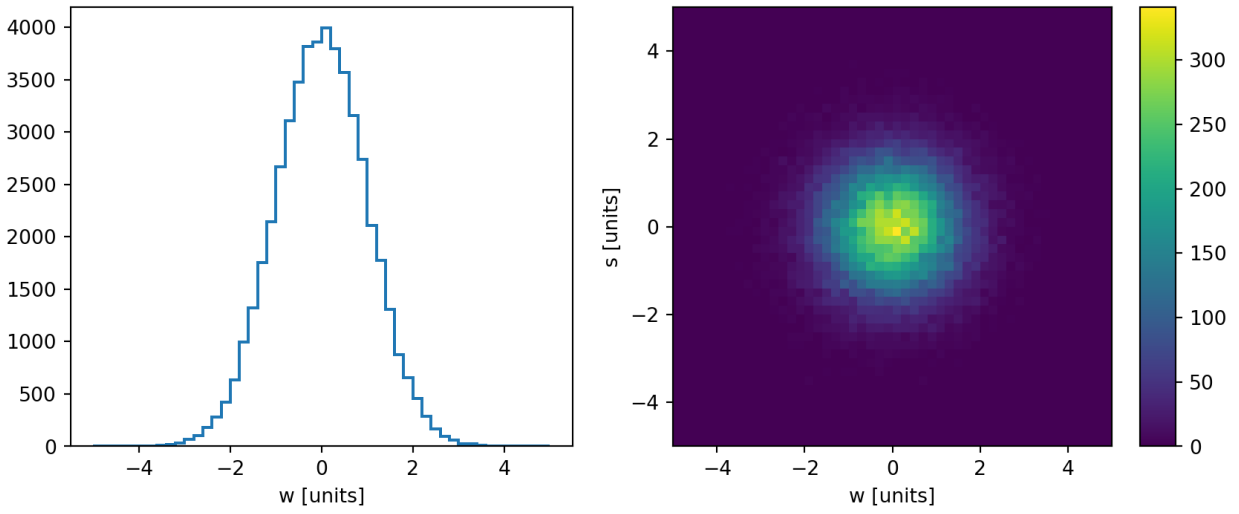
```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  storage=Double()) # (has staged fills)
```



[25]:

[26]:
```
# render in-memory histogram
h = h.compute()

import matplotlib.pyplot as plt

# auto-plot
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

h.project("W").plot(ax=axs[0])
h.project("W", "S").plot(ax=axs[1])
plt.show()
```

```
Warning:  you don't have flow bins stored in Hist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  storage=Double()) # Sum: 50000.0
```

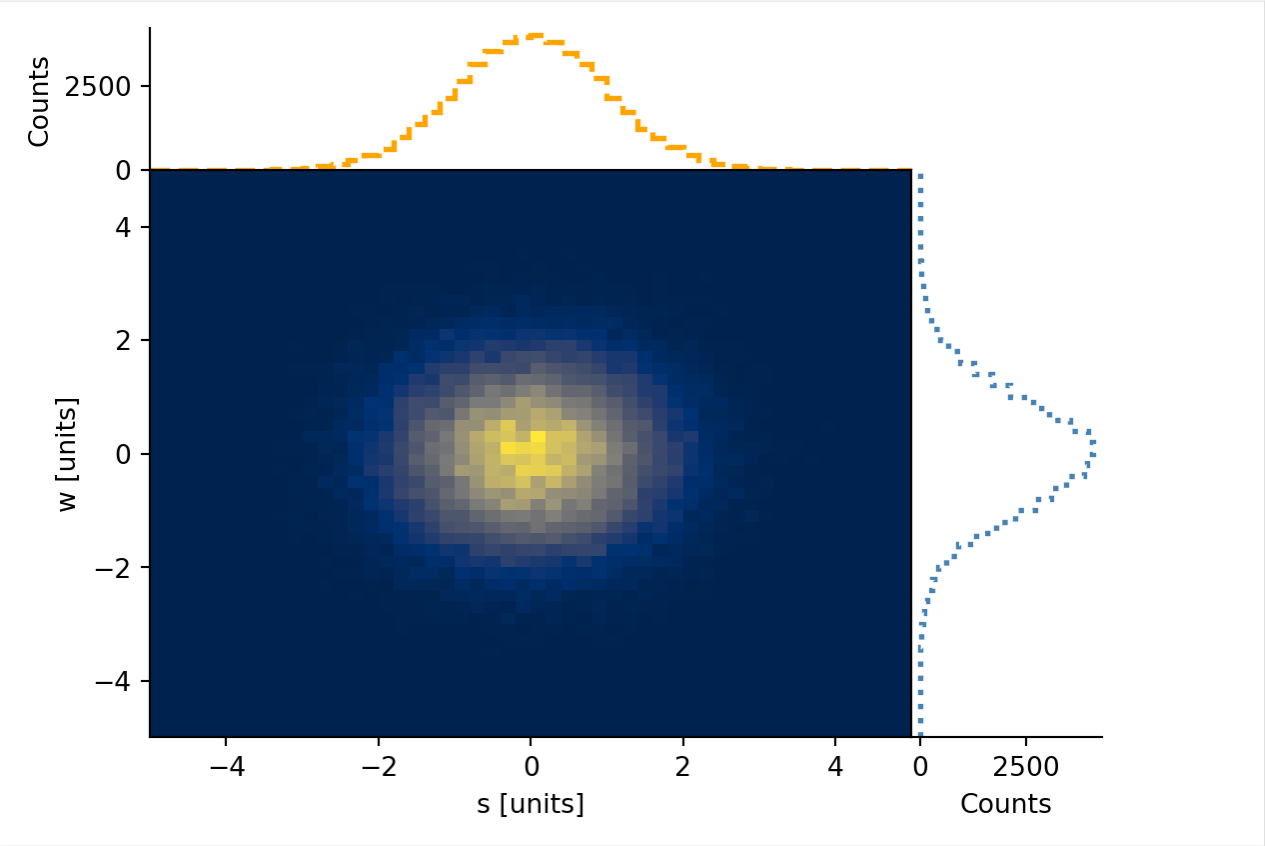


NamedHist

Below we'll use a dask array to fill a `hist.dask.NamedHist` lazily, as a proxy for filling it on a cluster, and then plot the resulting histogram!

[27]:
```
h = dah.NamedHist(
    hist.axis.Regular(
        50, -5, 5, name="S", label="s [units]", underflow=False, overflow=False
    ),
    hist.axis.Regular(
        50, -5, 5, name="W", label="w [units]", underflow=False, overflow=False
    ),
)
# should all use names
s_data = da.random.standard_normal(size=(50_000,), chunks=(1000,))
w_data = da.random.standard_normal(size=(50_000,), chunks=(1000,))

h.fill(W=w_data, S=s_data)

h = h.compute()
assert h[25, 25] == h[0j, 1j - 5] == h[{"W": 25, "S": 0j}]
assert h[:, 0:50:5j].project("S")

# plot2d full
h.plot2d_full(
    main_cmap="cividis",
    top_ls="--",
    top_color="orange",
    top_lw=2,
    side_ls=":",
    side_lw=2,
    side_color="steelblue",
)
plt.show()
```

```
Warning:  you don't have flow bins stored in NamedHist(
  Regular(50, -5, 5, underflow=False, overflow=False, name='S', label='s [units]'),
  Regular(50, -5, 5, underflow=False, overflow=False, name='W', label='w [units]'),
  storage=Double()) # Sum: 50000.0
```



1.14 Stack

1.14.1 Build via Axes

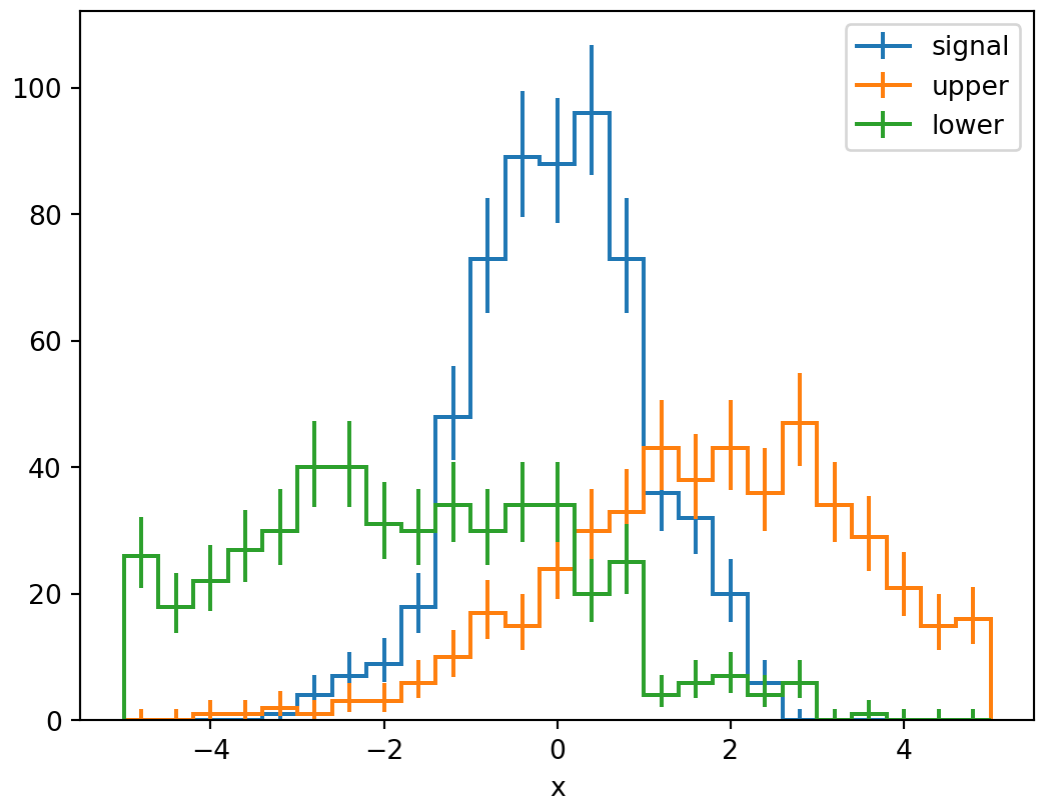
A histogram stack holds multiple 1-D histograms into a stack, whose axes are required to match. The most common way to create one is with a categorical axes:

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

import hist
from hist import Hist

ax = hist.axis.Regular(25, -5, 5, flow=False, name="x")
cax = hist.axis.StrCategory(["signal", "upper", "lower"], name="c")
full_hist = Hist(ax, cax)

full_hist.fill(x=np.random.normal(size=600), c="signal")
full_hist.fill(x=2 * np.random.normal(size=500) + 2, c="upper")
full_hist.fill(x=2 * np.random.normal(size=500) - 2, c="lower")


s = full_hist.stack("c")
```

You can build this directly with `hist.Stack(h1, h2, h3)`, `hist.Stack.from_iter([h1, h2, h3])`, or `hist.Stack.from_dict({"signal": h1, "lower": h2, "upper": h3})` as well.

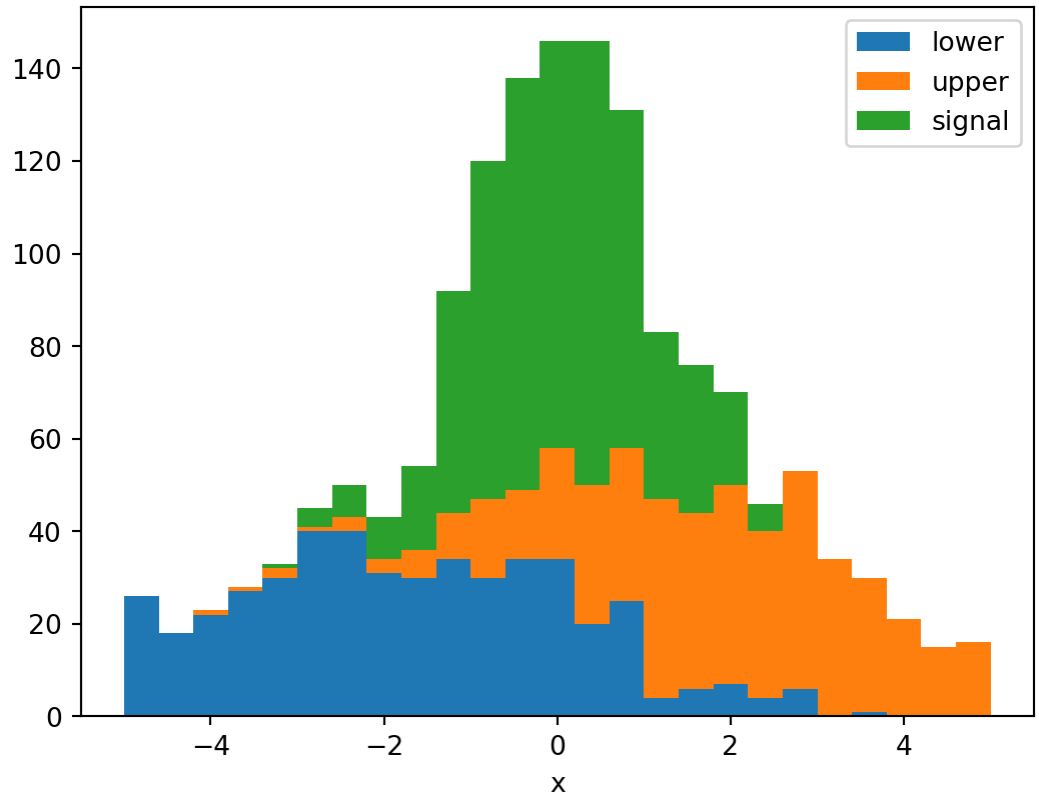
HistStack has `.plot()` method which calls mplhep and plots the histograms in the stack:

[2]:
```
s.plot()
plt.legend()
plt.show()
```



For the "stacked" style of plot, you can pass arguments through to mplhep. For some reason, this reverses the graphical order, but we can easily undo that by applying a slicing operation to the stack:

[3]:
```
s[::-1].plot(stack=True, histtype="fill")
plt.legend()
plt.show()
```



We can use `.show()` to access `histoprint` and print the stacked histograms to the console.

Note: Histoprint currently supports only non-discrete axes. Hence, it supports only regular and variable axes at the moment.

[4]:
```
s.show(columns=120)
```

```
 -5.000 _
↪                        146/row
 -4.600 _
 -4.200 _
 -3.800 _
 -3.400 _
 -3.000 _|
 -2.600 _
 -2.200 _||
 -1.800 _
 -1.400 _
 -1.000 _
 -0.600 _
 -0.200 _
  0.200 _
  0.600 _
  1.000 _
  1.400 _
  1.800 _
  2.200 _
```

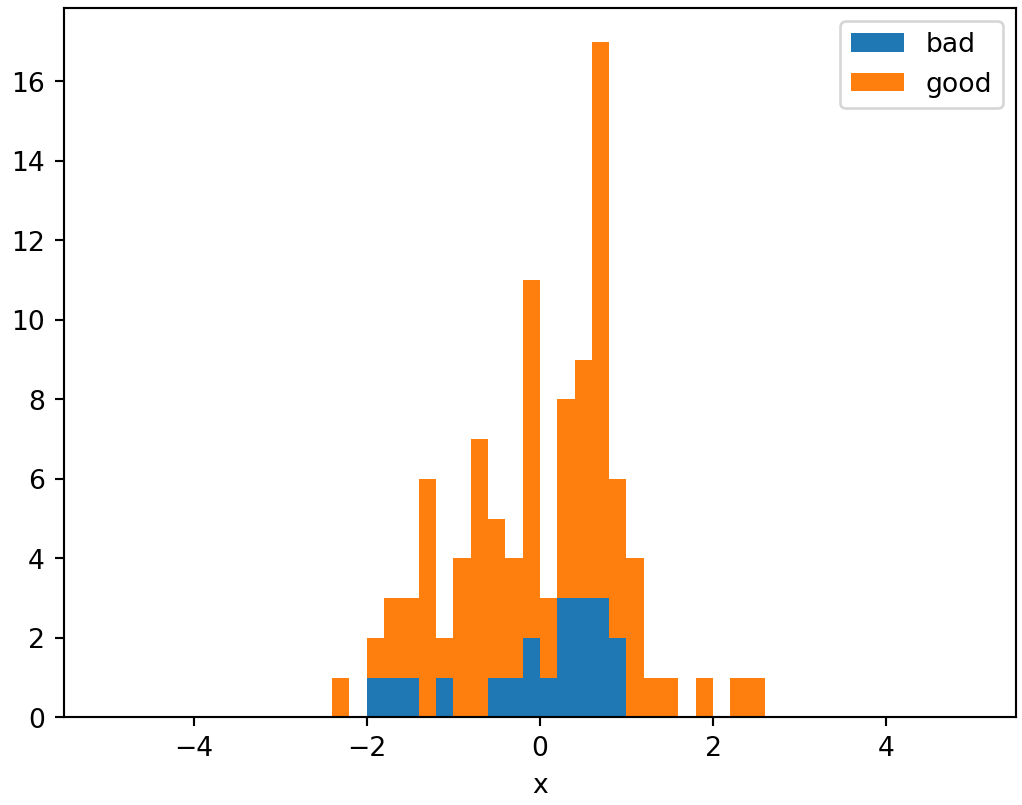
(continues on next page)

(continued from previous page)

```
 2.600 _
 3.000 _
 3.400 _
 3.800 _
 4.200 _
 4.600 _
 5.000 _
                                        signal    | upper      lower
```

1.14.2 Manipulations on a Stack

[5]:
```
h = Hist.new.Reg(50, -5, 5, name="x").StrCat(["good", "bad"], name="quality").Double()

h.fill(x=np.random.randn(100), quality=["good", "good", "good", "good", "bad"] * 20)

# Turn an existing axis into a stack
s = h.stack("quality")
s[::-1].plot(stack=True, histtype="fill")
plt.legend()
plt.show()
```



Histograms in a stack can have names. The names of histograms are the categories, which are corresponding profiled histograms:

[6]:
```
print(s[0].name)
s[0]
```

```
good
```

[6]:
```
Hist(Regular(50, -5, 5, name='x'), storage=Double()) # Sum: 80.0
```

You can use those names in indexing, just like for axes (only when using string names):

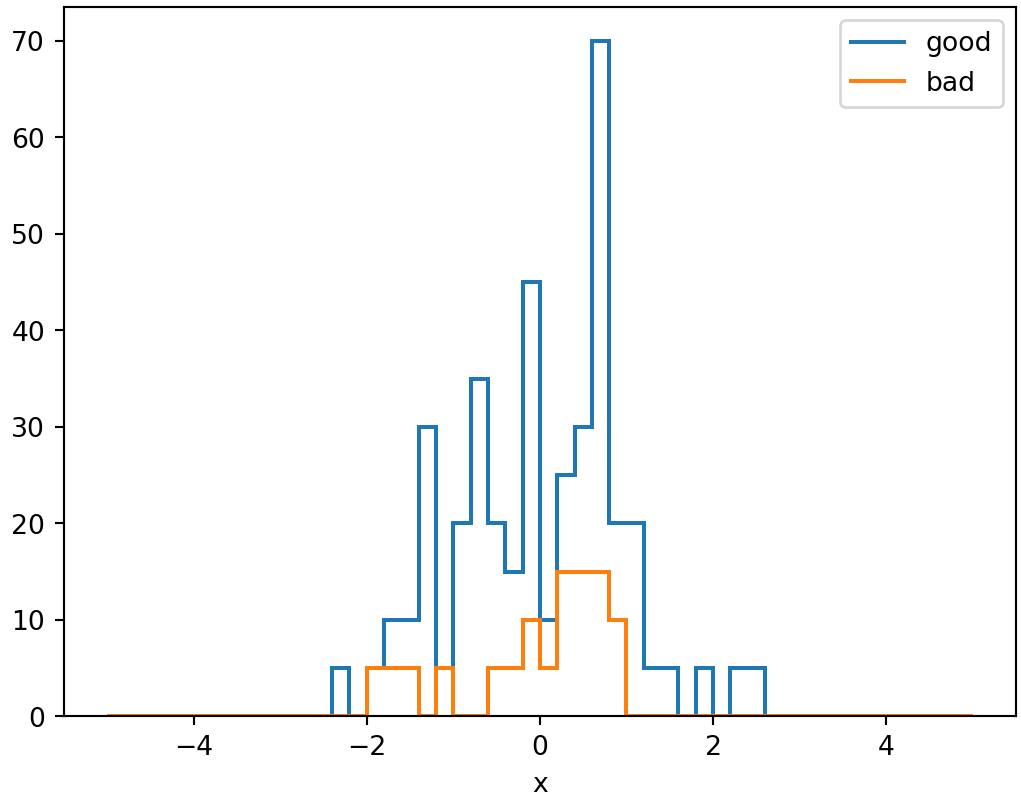
[7]:
```
print(s["bad"].name)
s["bad"]
```

```
bad
```

[7]:
```
Hist(Regular(50, -5, 5, name='x'), storage=Double()) # Sum: 20.0
```

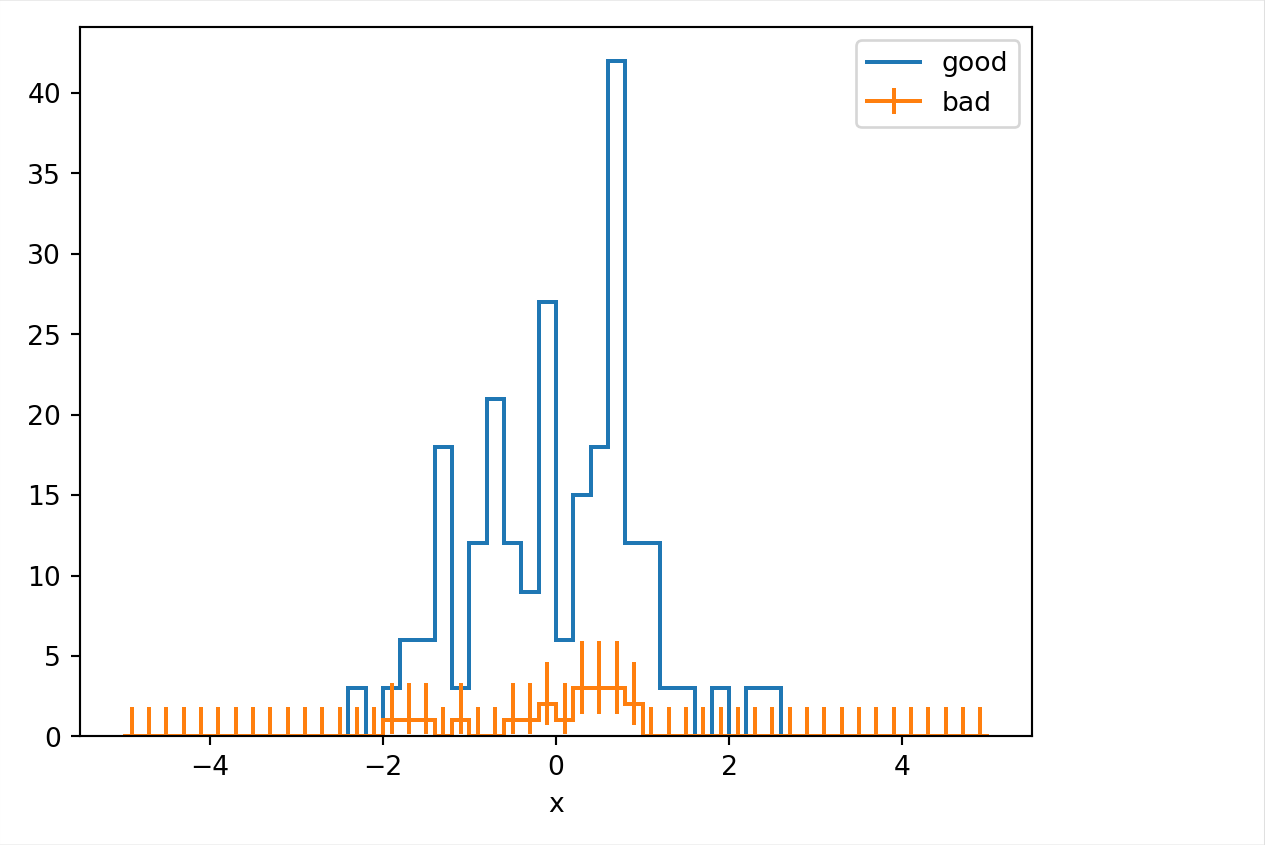
You can scale a stack:

[8]:
```
(s * 5).plot()
plt.legend()
plt.show()
```



Or an item in the stack inplace:

[9]:
```
s["good"] *= 3
s.plot()
plt.legend()
plt.show()
```



You can project on a stack, as well, if the histograms are at least two dimensional. `h.stack("x").project("y")` is identical to `h.project("y").stack("x")`.

1.15 Interpolation

1.15.1 Via SciPy

We can perform interpolation in Hist using SciPy.

[1]:
```
# Make the necessary imports.
import matplotlib.pyplot as plt
import numpy as np
from scipy import interpolate

from hist import Hist
```

[2]:
```
# We obtain evenly spaced numbers over the specified interval.
x = np.linspace(-27, 27, num=250, endpoint=True)

# Define a Hist object and fill it.
h = Hist.new.Reg(10, -30, 30).Double()
centers = h.axes[0].centers
```

(continues on next page)

(continued from previous page)

```
weights = np.cos(-(centers**2) / 9.0) ** 2
h.fill(centers, weight=weights)
```

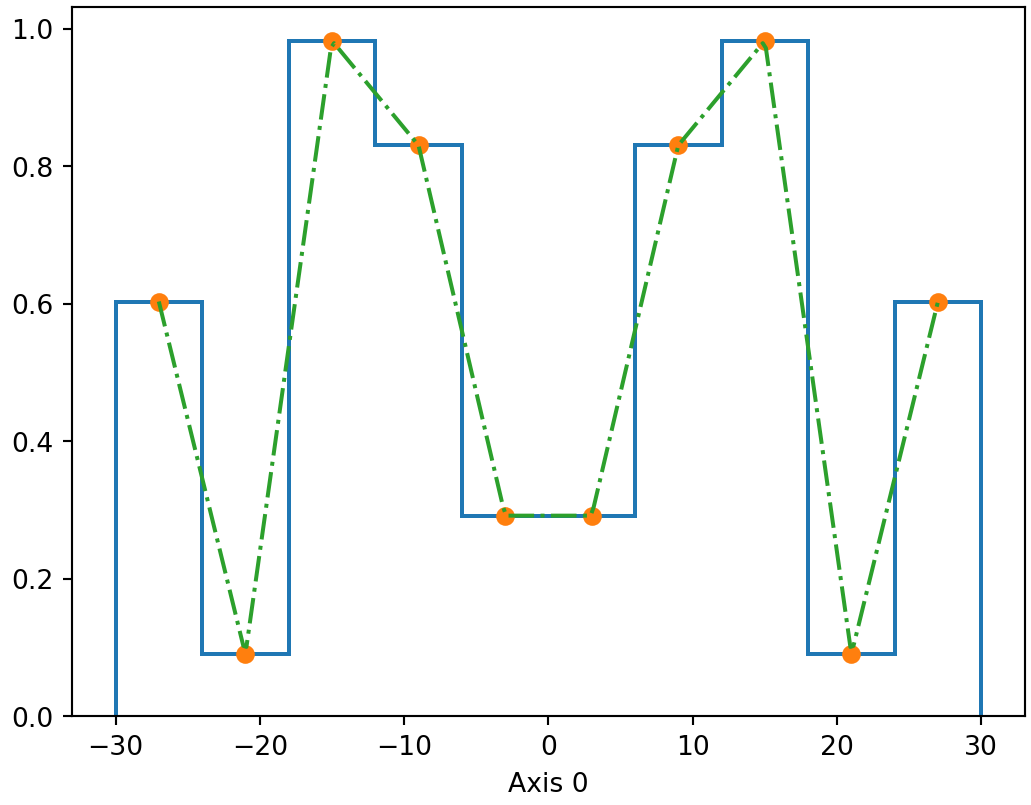
[2]:
```
Hist(Regular(10, -30, 30, label='Axis 0'), storage=Double()) # Sum: 5.596329884235402
```

Linear 1-D

We can obtain a linear interpolation by passion the `kind="linear"` argument in interpolate.interp1d().

[3]:
```
linear_interp = interpolate.interp1d(h.axes[0].centers, h.values(), kind="linear")
```

[4]:
```
h.plot()  # Plot the histogram
plt.plot(h.axes[0].centers, h.values(), "o")  # Mark the bin centers
plt.plot(x, linear_interp(x), "-.")  # Plot the Linear interpolation
plt.show()
```

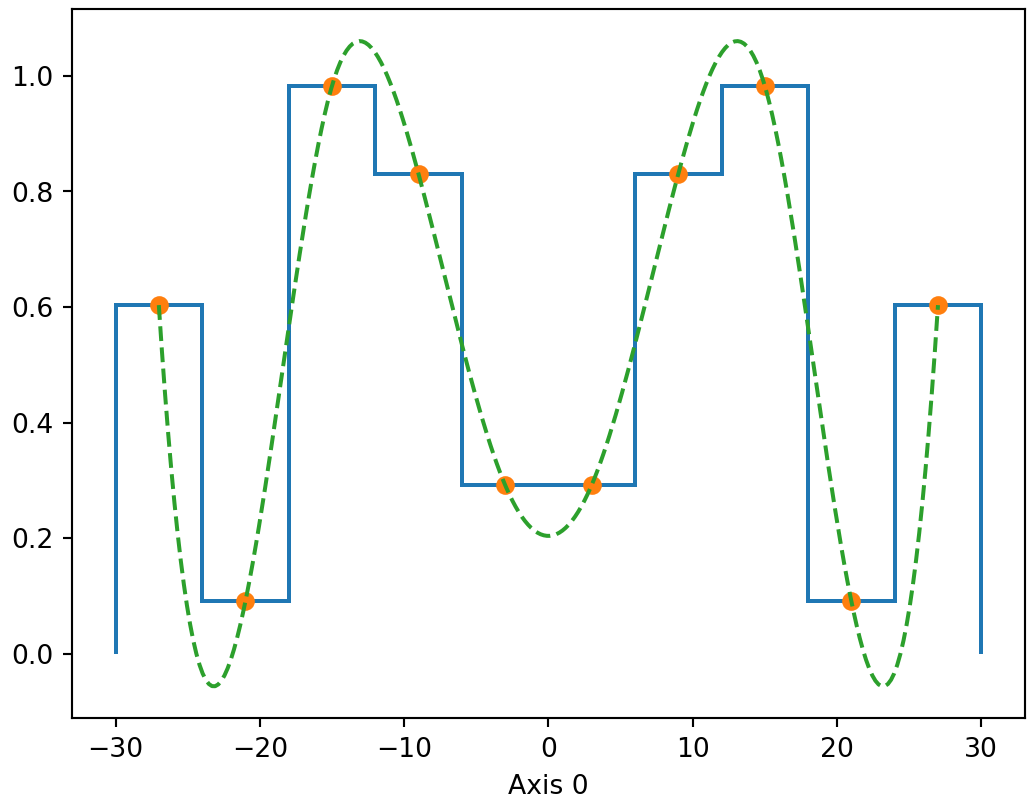


Cubic 1-D

We can obtain a cubic interpolation by passion the `kind="cubic"` argument in interpolate.interp1d().

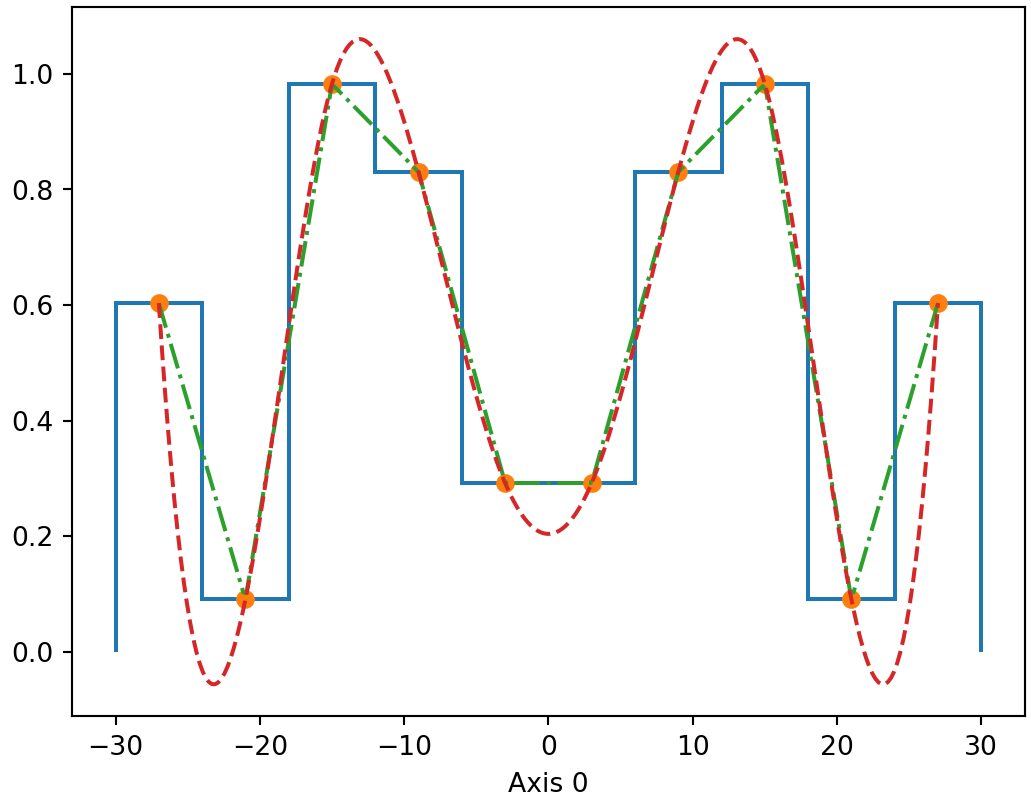
[5]:
```
cubic_interp = interpolate.interp1d(h.axes[0].centers, h.values(), kind="cubic")
```

[6]:
```
h.plot()  # Plot the histogram
plt.plot(h.axes[0].centers, h.values(), "o")  # Mark the bin centers
plt.plot(x, cubic_interp(x), "--")  # Plot the Cubic interpolation
plt.show()
```



We can also plot them both together to compare the interpolations.

[7]:
```
h.plot()  # Plot the histogram
plt.plot(h.axes[0].centers, h.values(), "o")  # Mark the bin centers
plt.plot(x, linear_interp(x), "-.")  # Plot the Linear interpolation
plt.plot(x, cubic_interp(x), "--")  # Plot the Cubic interpolation
plt.show()
```



1.16 CLI

For compatibility with the original Hist library, we provide a CLI as well.

```
usage: hist [-h] [-i INPUT] [-b BUCKETS] [-s SCREEN_WIDTH] [-t LABEL]
            [-o OUTPUT_IMAGE]

options:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        input file to read from (stdin by default)
  -b BUCKETS, --buckets BUCKETS
                        number of bins
  -s SCREEN_WIDTH, --screen-width SCREEN_WIDTH
                        maximum screen width
  -t LABEL, --label LABEL
                        label for plot
  -o OUTPUT_IMAGE, --output-image OUTPUT_IMAGE
                        save image to file
```

See the Scikit-HEP Developer introduction for a detailed description of best practices for developing Scikit-HEP packages.

1.17 Contributing

1.17.1 Setting up a development environment

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 with `pip install pipx` (the only case were 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 build  # Make an SDist and wheel
```

Nox handles everything for you, including setting up an temporary virtual environment for each run. On Linux, it will run the `--mpl` tests. You can run the linux tests from anywhere with Docker:

```
docker run --rm -v $PWD:/nox -w /nox -t quay.io/pypa/manylinux2014_x86_64:latest pipx␣
↪run nox -s tests-3.9
# Regenerate the MPL comparison images:
docker run --rm -v $PWD:/nox -w /nox -t quay.io/pypa/manylinux2014_x86_64:latest pipx␣
↪run nox -s regenerate
```

PyPI

For extended development, you can set up a development environment using PyPI.

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv)$ pip install -e .[dev]
(venv)$ python -m ipykernel install --user --name hist
```

You should have pip 10 or later.

Conda

You can also set up a development environment using Conda. With conda, you can search some channels for development.

```
$ conda env create -f dev-environment.yml -n hist
$ conda activate hist
(hist)$ python -m ipykernel install --name hist
```

1.17.2 Post setup

You should prepare pre-commit, which will help you by checking that commits pass required checks:

```
pipx install pre-commit # or brew install pre-commit on macOS
pre-commit install # Will install a pre-commit hook into the git repo
```

You can also/alternatively run `pre-commit run` (changes only) or `pre-commit run --all-files` to check even without installing the hook.

1.17.3 Testing

Use PyTest to run the unit checks:

```
pytest
```

1.18 Support

If you are stuck with a problem using Hist, 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.

1.19 Changelog

1.19.1 Version 2.7.2

- Support boost-histogram 1.4.0 in addition to 1.3.x, including Python 3.12, flow disabling for categories, and integer arrays required for integer axes #535 and #532
- Add a `.T` shortcut #521
- Support ND quick constructs being mistakenly passed in #528
- Add `boost` nox job to test upstream changes easily #533

1.19.2 Version 2.7.1

- `[plot]` extra split into `[fit]` #515
- Use PyPI trusted publisher to deploy #516

1.19.3 Version 2.7.0

Features:

- Add a function to integrate axes #505
- Add `fill_flattened` support #474

Various other items:

- Move linting to using Ruff #475
- `Self` & `_compat` #479
- Rework and rerun docs API #481

1.19.4 Version 2.6.3

- Experimental dask support #471
- Some minor cleanups from refurb #453

1.19.5 Version 2.6.2

- Nicer stacks repr #449
- Backport `storage_type` if boost-histogram < 1.3.2 #447
- Allow overwriting labels for plot/overlay #414
- Use Hatching to build the package #418
- Support git archival version numbers #441

1.19.6 Version 2.6.1

- Fall back on normal repr when histogram is too large #388
- Fix issue with no-axis histogram #388
- Fix issue with empty axis causing segfault until fixed upstream #387
- Only require SciPy if using SciPy #386

1.19.7 Version 2.6.0

- Using boost-histogram 1.3
- Fix runtime dependency on matplotlib when not plotting #353
- Fix `.plot` shortcut failure #368
- New nox sessions: `regenerate` and `pylint`
- Update tests for latest matplotlib

1.19.8 Version 2.5.2

- Remove more-itertools requirement #347
- Fix missing pass-through for stack plot #339

1.19.9 Version 2.5.1

- Support named stack indexing #325
- Fix histoprint error with stacks #325
- Better README

1.19.10 Version 2.5.0

- Stacks support axes, math operations, projection, setting items, and iter/dict construction. They also support histogram titles in legends. Added histoprint support for Stacks. #291 #315 #317 #318
- Added `name=` and `label=` to histograms, include Hist arguments in QuickConstruct. #297
- AxesTuple now supports bulk name setting, `h.axes.name = ("a", "b", ...)`. #288
- Added `hist.new` alias for `hist.Hist.new`. #296
- Added `"efficiency"` `uncertainty_type` option for `ratio_plot` API. #266 #278

Smaller features or fixes:

- Dropped Python 3.6 support. #194
- Uses boost-histogram 1.2.x series, includes all features and fixes, and Python 3.10 support.
- No longer require scipy or iminuit unless actually needed. #316
- Improve and clarify treatment of confidence intervals in `intervals` submodule. #281
- Use NumPy 1.21 for static typing. #285
- Support running tests without plotting requirements. #321

1.19.11 Version 2.4.0

- Support `.stack(axis)` and stacked histograms. #244 #257 #258
- Support selection lists (experimental with boost-histogram 1.1.0). #255
- Support full names for QuickConstruct, and support mistaken usage in constructor. #256
- Add `.sort(axis)` for quickly sorting a categorical axis. #243

Smaller features or fixes:

- Support nox for easier contributor setup. #228
- Better name axis error. #232
- Fix for issue plotting size 0 axes. #238
- Fix issues with repr information missing. #241
- Fix issues with wrong plot shortcut being triggered by Integer axes. #247
- Warn and better error if overlapping keyword used as axis name. #250

Along with lots of smaller docs updates.

1.19.12 Version 2.3.0

- Add `plot_ratio` to the public API, which allows for making ratio plots between the histogram and either another histogram or a callable. #161
- Add `.profile` to compute a (N-1)D profile histogram. #160
- Support plot1d / plot on Histograms with a categorical axis. #174
- Add frequentist coverage interval support in the `intervals` module. #176
- Allow `plot_pull` to take a more generic callable or a string as a fitting function. Introduce an option to perform a likelihood fit. Write fit parameters' values and uncertainties in the legend. #149
- Add `fit_fmt=` to `plot_pull` to control display of fit params. #168
- Support `<prefix>_kw` arguments for setting each axis in plotting. #193
- Cleaner IPython completion for Python 3.7+. #179

1.19.13 Version 2.2.1

- Fix bug with `plot_pull` missing a sqrt. #150
- Fix static typing with ellipses. #145
- Require boost-histogram 1.0.1+, fixing typing related issues, allowing subclassing Hist without a family and including a important Mean/WeighedMean summing fix. #151

1.19.14 Version 2.2.0

- Support boost-histogram 1.0. Better plain reprs. Full Static Typing. #137
- Support `data=` when construction a histogram to copy in initial data. #142
- Support `Hist.from_columns`, for simple conversion of DataFrames and similar structures #140
- Support `.plot_pie` for quick pie plots #140

1.19.15 Version 2.1.1

- Fix density (and density based previews) #134

1.19.16 Version 2.1.0

- Support shortcuts for setting storages by string or position #129

Updated dependencies:

- `boost-histogram` 0.11.0 to 0.13.0.
  - Major new features, including PlottableProtocol
- `histoprint` >=1.4 to >=1.6.
- `mplhep` >=0.2.16 when `[plot]` given

1.19.17 Version 2.0.1

- Make sum of bins explicit in notebook representations. #106
- Fixed `plot2d_full` incorrectly mirroring the y-axis. #105
- `Hist.plot_pull`: more suitable bands in the pull bands 1sigma, 2 sigma, etc. #102
- Fixed classichist's usage of `get_terminal_size` to support not running in a terminal #99

1.19.18 Version 2.0.0

First release of Hist.

1.20 Hist Quick Demo

My favorite demo notebook config setting:

[1]:
```
%config InteractiveShell.ast_node_interactivity="last_expr_or_assign"
```

Let's import Hist:

[2]:
```
import numpy as np

from hist import Hist
```

We can use the classic constructors from boost-histogram, but let's use the new QuickConstruct system instead:

```
[3]: h = Hist.new.Reg(100, -10, 10, name="x").Double()
```

```
[3]: Hist(Regular(100, -10, 10, name='x'), storage=Double())
```

Let's fill it with some data:

```
[4]: h.fill(np.random.normal(1, 3, 1_000_000))
```

```
[4]: Hist(Regular(100, -10, 10, name='x'), storage=Double()) # Sum: 998531.0 (1000000.0 with␣
↪flow)
```

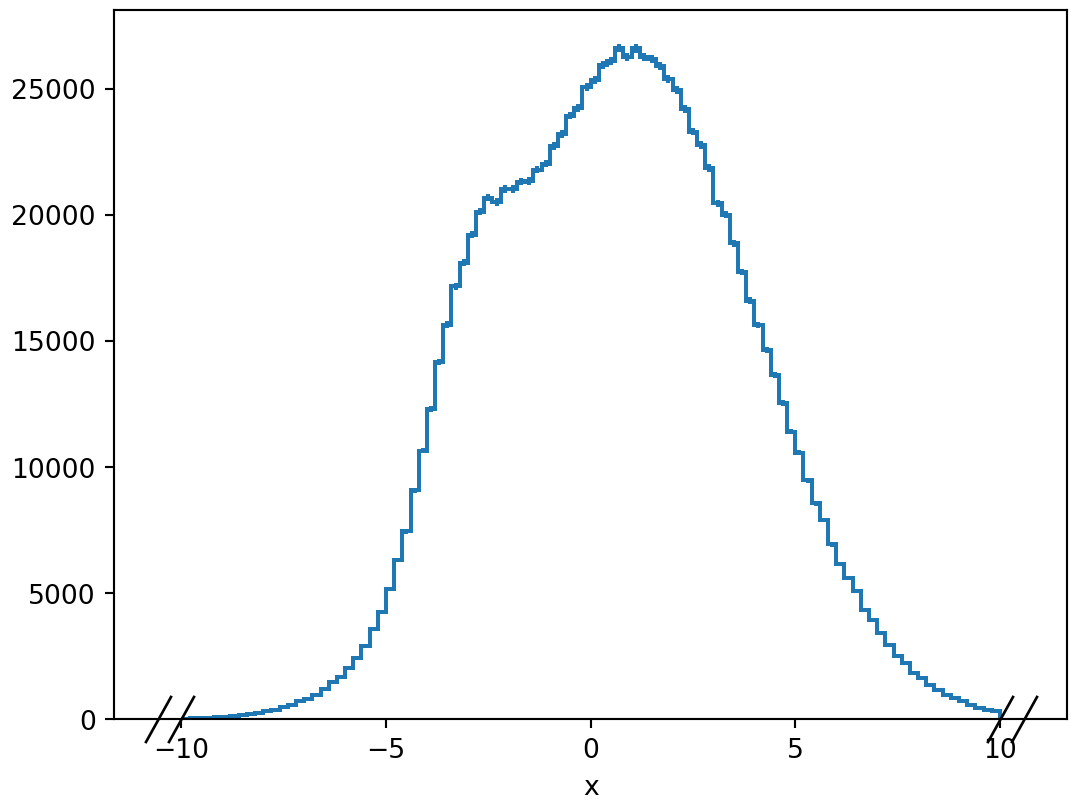
And you can keep filling:

```
[5]: h.fill(np.random.normal(-3, 1, 100_000))
```

```
[5]: Hist(Regular(100, -10, 10, name='x'), storage=Double()) # Sum: 1098531.0 (1100000.0 with␣
↪flow)
```

You can plot (uses mplhep in the backend):

```
[6]: h.plot();
```



We also have direct access to histoprint:

```
[7]: h.show(columns=50)
```

```
-1.000 _ x 10^+01                    26620/row
-0.980 _
-0.960 _
```

(continues on next page)

(continued from previous page)

```
-0.940 _
-0.920 _
-0.900 _
-0.880 _
-0.860 _
-0.840 _
-0.820 _
-0.800 _
-0.780 _
-0.760 _
-0.740 _
-0.720 _
-0.700 _
-0.680 _
-0.660 _
-0.640 _
-0.620 _
-0.600 _
-0.580 _
-0.560 _
-0.540 _
-0.520 _
-0.500 _
-0.480 _
-0.460 _
-0.440 _
-0.420 _
-0.400 _
-0.380 _
-0.360 _
-0.340 _
-0.320 _
-0.300 _
-0.280 _
-0.260 _
-0.240 _
-0.220 _
-0.200 _
-0.180 _
-0.160 _
-0.140 _
-0.120 _
-0.100 _
-0.080 _
-0.060 _
-0.040 _
-0.020 _
 0.000 _
 0.020 _
 0.040 _
 0.060 _
 0.080 _
```

(continues on next page)

(continued from previous page)

```
  0.100 _
  0.120 _
  0.140 _
  0.160 _
  0.180 _
  0.200 _
  0.220 _
  0.240 _
  0.260 _
  0.280 _
  0.300 _
  0.320 _
  0.340 _
  0.360 _
  0.380 _
  0.400 _
  0.420 _
  0.440 _
  0.460 _
  0.480 _
  0.500 _
  0.520 _
  0.540 _
  0.560 _
  0.580 _
  0.600 _
  0.620 _
  0.640 _
  0.660 _
  0.680 _
  0.700 _
  0.720 _
  0.740 _
  0.760 _
  0.780 _
  0.800 _
  0.820 _
  0.840 _
  0.860 _
  0.880 _
  0.900 _
  0.920 _
  0.940 _
  0.960 _
  0.980 _
  1.000 _
```

Let's try 2D:

[8]:
```
h2 = Hist.new.Reg(100, -10, 10, name="x").Reg(100, -10, 10, name="y").Double()
```

[8]:
```
Hist(
  Regular(100, -10, 10, name='x'),
```

(continues on next page)

(continued from previous page)

```
  Regular(100, -10, 10, name='y'),
  storage=Double())
```

Can fill with two arrays:

[9]:
```
h2.fill(x=np.random.normal(-3, 2, 500_000), y=np.random.normal(3, 1, 500_000))
```

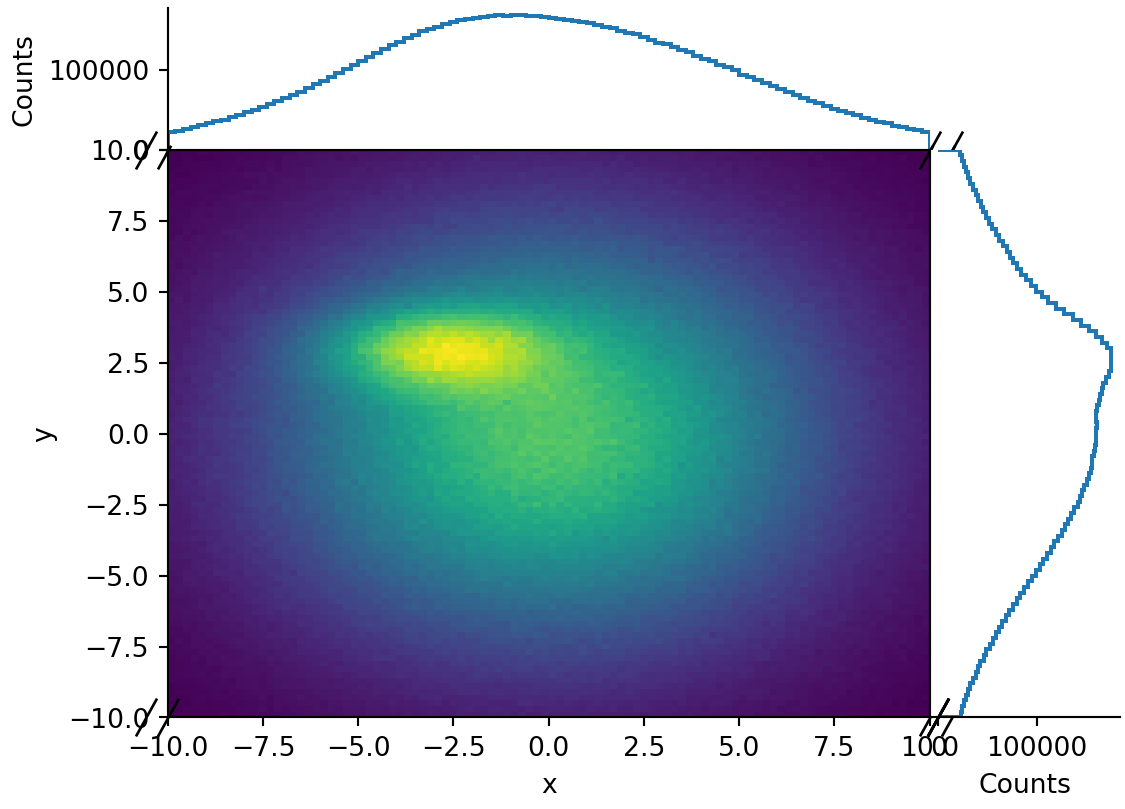
[9]:
```
Hist(
  Regular(100, -10, 10, name='x'),
  Regular(100, -10, 10, name='y'),
  storage=Double()) # Sum: 499889.0 (500000.0 with flow)
```

Or a 2D array (hey, let's do a multithreaded fill just for fun, too!):

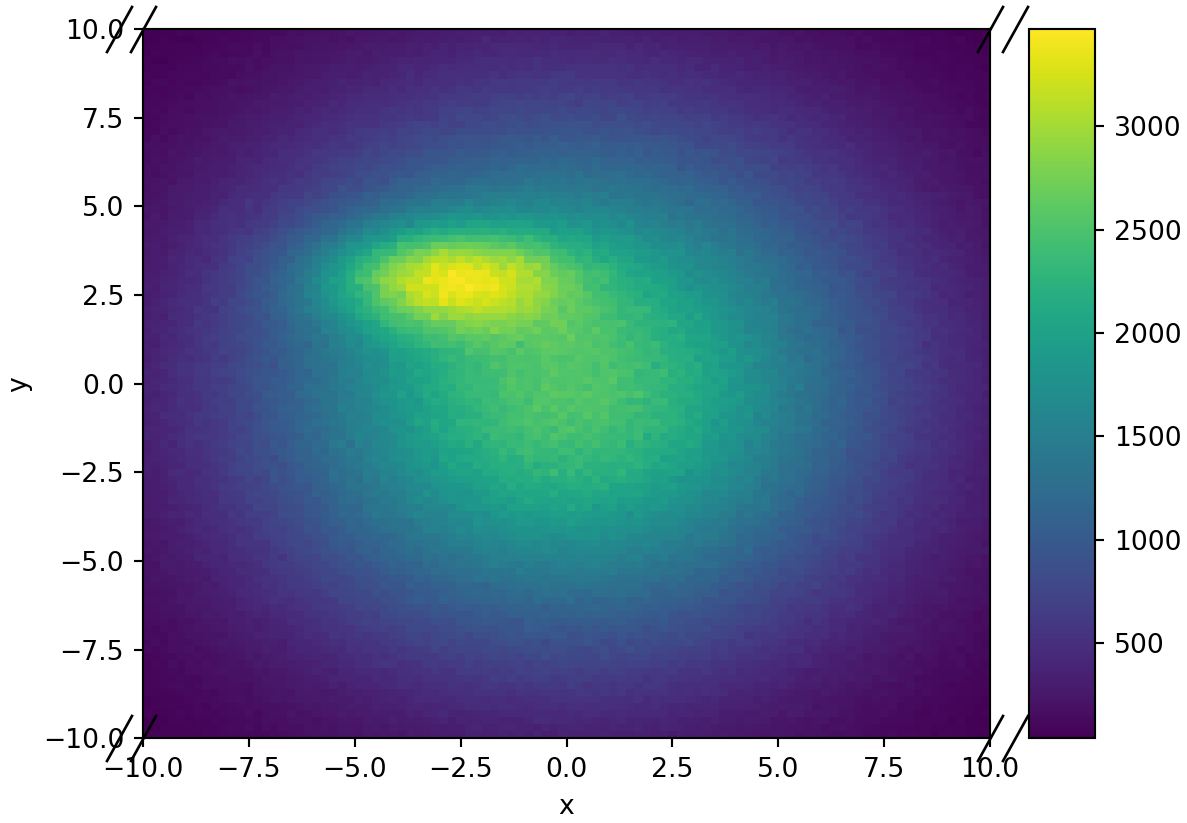
[10]:
```
h2.fill(*np.random.normal(0, 5, (2, 10_000_000)), threads=4)
```

[10]:
```
Hist(
  Regular(100, -10, 10, name='x'),
  Regular(100, -10, 10, name='y'),
  storage=Double()) # Sum: 9611675.0 (10500000.0 with flow)
```

[11]:
```
h2.plot2d_full();
```



[12]:
```
h2.plot();
```



[13]:
```
h2.project("x")
```

[13]:
```
Hist(Regular(100, -10, 10, name='x'), storage=Double()) # Sum: 10046154.0 (10500000.0
↪with flow)
```

[14]:
```
h3 = (
    Hist.new.Reg(100, -10, 10, name="x")
    .Reg(50, -5, 5, name="y")
    .Reg(60, -3, 3, name="z")
    .Double()
)
```

[14]:
```
Hist(
  Regular(100, -10, 10, name='x'),
  Regular(50, -5, 5, name='y'),
  Regular(60, -3, 3, name='z'),
  storage=Double())
```

[15]:
```
h3.fill(*np.random.normal(0, 5, (3, 10_000_000)))
```

[15]:
```
Hist(
  Regular(100, -10, 10, name='x'),
  Regular(50, -5, 5, name='y'),
  Regular(60, -3, 3, name='z'),
  storage=Double()) # Sum: 2941844.0 (10000000.0 with flow)
```

[16]:
```
h3.project("x", "y")
# Can also write:
# h3[:, :, sum]
# h3[..., sum]
```

[16]:
```
Hist(
  Regular(100, -10, 10, name='x'),
  Regular(50, -5, 5, name='y'),
  storage=Double()) # Sum: 6516536.0 (10000000.0 with flow)
```

We can slice and dice. Plain numbers refer to bins. Use a "j" suffix to refer to data coordinates. As above, `sum` will sum over an axis (optionally with end points). This system is called UHI+.

[17]:
```
h3[-8j:8j, 10:50, sum]
```

[17]:
```
Hist(
  Regular(80, -8, 8, name='x'),
  Regular(40, -3, 5, name='y'),
  storage=Double()) # Sum: 5049851.0 (10000000.0 with flow)
```

You can also use a dict; that includes using names too. (Note: this was independently developed but is nearly identical to XArray)

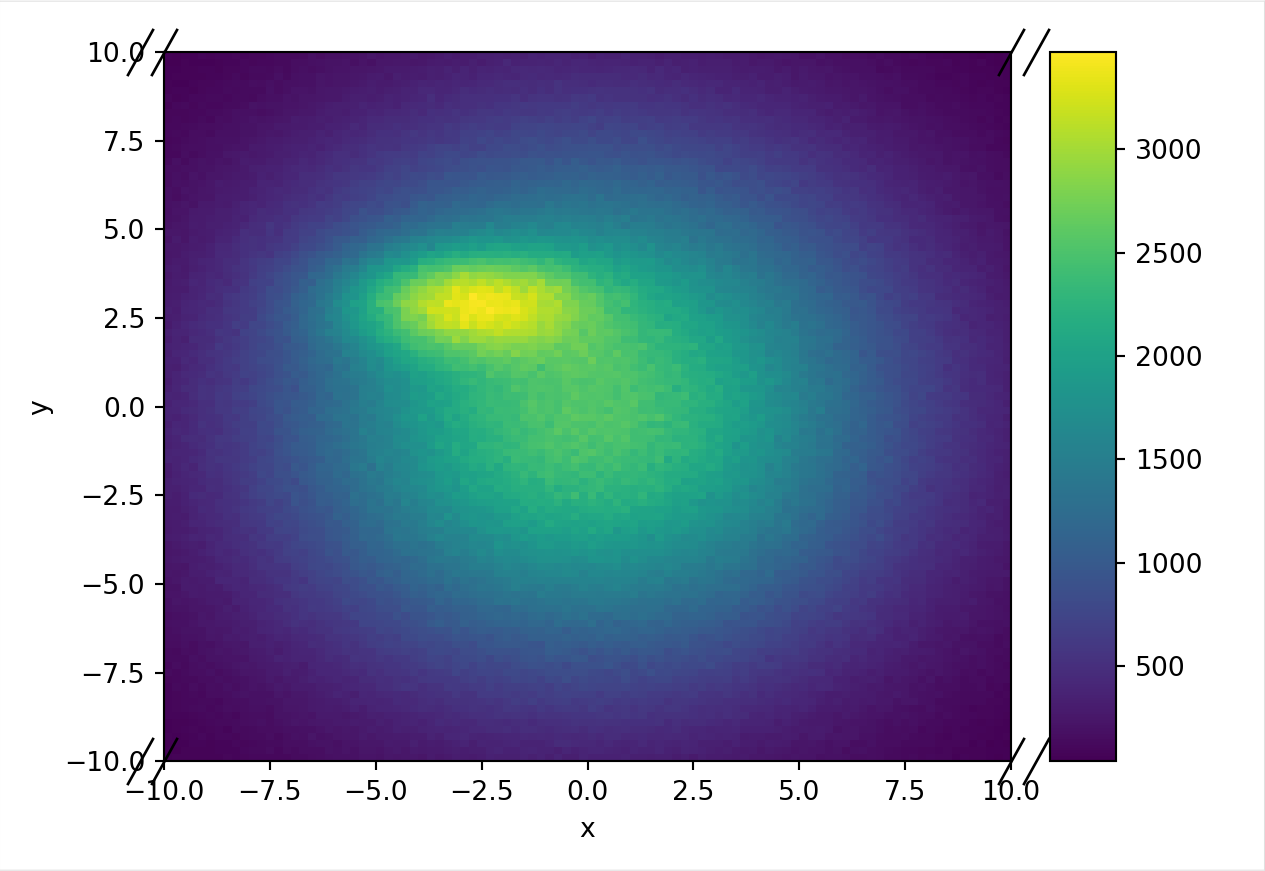
[18]:
```
h3[{"x": slice(-8j, 8j), "y": slice(10, 50), "z": sum}]
```

[18]:
```
Hist(
  Regular(80, -8, 8, name='x'),
  Regular(40, -3, 5, name='y'),
  storage=Double()) # Sum: 5049851.0 (10000000.0 with flow)
```

Everything integrates with histoprint, uproot4, and mplhep, too:

[19]:
```
import mplhep
```

[20]:
```
mplhep.hist2dplot(h2);
```



1.21 hist

1.21.1 hist package

class hist.**BaseHist**(**in_args: AxisTypes | Storage | str*, *storage: Storage | str | None = None*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*)

Bases: `Histogram`

property T: Self

density() → np.typing.NDArray[Any]

Density NumPy array.

fill(**args: Any*, *weight: Any | None = None*, *sample: Any | None = None*, *threads: int | None = None*, ***kwargs: Any*) → Self

Insert data into the histogram using names and indices, return a Hist object.

fill_flattened(**args: Any*, *weight: Any | None = None*, *sample: Any | None = None*, *threads: int | None = None*, ***kwargs: Any*) → Self

classmethod from_columns(*data: Mapping[str, Any]*, *axes: Sequence[str | AxisProtocol]*, ***, *weight: str | None = None*, *storage: Storage = Double()*) → Self

integrate(*name: int | str*, *i_or_list: InnerIndexing | list[str | int] | None = None*, *j: InnerIndexing | None = None*) → Self | float | bh.accumulators.Accumulator

plot(**args: Any*, *overlay: str | None = None*, ***kwargs: Any*) → Hist1DArtists | Hist2DArtists

Plot method for BaseHist object.

plot1d(*, *ax: matplotlib.axes.Axes | None = None*, *overlay: str | int | None = None*, ***kwargs: Any*) → Hist1DArtists

Plot1d method for BaseHist object.

plot2d(*, *ax: matplotlib.axes.Axes | None = None*, ***kwargs: Any*) → Hist2DArtists

Plot2d method for BaseHist object.

plot2d_full(*, *ax_dict: dict[str, matplotlib.axes.Axes] | None = None*, ***kwargs: Any*) → tuple[Hist2DArtists, Hist1DArtists, Hist1DArtists]

Plot2d_full method for BaseHist object.

Pass a dict of axes to `ax_dict`, otherwise, the current figure will be used.

plot_pie(*, *ax: matplotlib.axes.Axes | None = None*, ***kwargs: Any*) → Any

plot_pull(*func: Callable[[np.typing.NDArray[Any]], np.typing.NDArray[Any]] | str*, *, *ax_dict: dict[str, matplotlib.axes.Axes] | None = None*, ***kwargs: Any*) → tuple[FitResultArtists, RatiolikeArtists]

`plot_pull` method for `BaseHist` object.

Return a tuple of artists following a structure of `(main_ax_artists, subplot_ax_artists)`

plot_ratio(*other:* hist.BaseHist | *Callable[[np.typing.NDArray[Any]], np.typing.NDArray[Any]] | str*, *, *ax_dict: dict[str, matplotlib.axes.Axes] | None = None*, ***kwargs: Any*) → tuple[MainAxisArtists, RatiolikeArtists]

`plot_ratio` method for `BaseHist` object.

Return a tuple of artists following a structure of `(main_ax_artists, subplot_ax_artists)`

profile(*axis: int | str*) → Self

Returns a profile (Mean/WeightedMean) histogram from a normal histogram with N-1 axes. The axis given is profiled over and removed from the final histogram.

project(**args: int | str*) → Self | float | Any

Projection of axis idx.

show(***kwargs: Any*) → Any

Pretty print histograms to the console.

sort(*axis: int | str*, *key: Callable[[int],* SupportsLessThan*] | Callable[[str],* SupportsLessThan*] | None = None*, *reverse: bool = False*) → Self

Sort a categorical axis.

stack(*axis: int | str*) → *Stack*

Returns a stack from a normal histogram axes.

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

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

class hist.**Hist**(**in_args: AxisTypes |* Storage *| str*, *storage:* Storage *| str | None = None*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*)

Bases: *BaseHist*

class hist.**NamedHist**(**args: Any*, ***kwargs: Any*)

Bases: *BaseHist*

fill(*weight: Any | None = None*, *sample: Any | None = None*, *threads: int | None = None*, ***kwargs: Any*) → Self

Insert data into the histogram using names and return a NamedHist object. NamedHist could only be filled by names.

fill_flattened(*obj: Any | None = None*, ***, *weight: Any | None = None*, *sample: Any | None = None*, *threads: int | None = None*, ***kwargs: Any*) → Self

project(**args: int | str*) → Self | float | Any

Projection of axis idx.

class hist.**Stack**(**args:* BaseHist)

Bases: object

property **axes**: *NamedAxesTuple*

classmethod **from_dict**(*d: Mapping[str,* BaseHist*]*) → Self

Create a Stack from a dictionary of histograms. The keys of the dictionary are used as names.

classmethod **from_iter**(*iterable: Iterable[*BaseHist*]*) → Self

Create a Stack from an iterable of histograms.

plot(***, *ax: mpl.axes.Axes | None = None*, ***kwargs: Any*) → Any

Plot method for Stack object.

project(**args: int | str*) → Self

Project the Stack onto a new axes.

show(***kwargs: object*) → Any

Pretty print the stacked histograms to the console.

class hist.**loc**(*value: str | float*, *offset: int = 0*)

Bases: *Locator*

value

class hist.**rebin**(*value: int*)

Bases: object

factor

hist.**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.

Subpackages

hist.axis package

class hist.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 hist.axis.**AxesMixin**

Bases: object

property label: str

Get or set the label for the Regular axis

property name: str

Get the name for the Regular axis

class hist.axis.**AxisProtocol**(**args*, ***kwargs*)

Bases: Protocol

label: str

metadata: Any

property name: str

class hist.axis.**Boolean**(***, *name: str = ''*, *label: str = ''*, *metadata: Any | None = None*, *__dict__: dict[str, Any] | None = None*)

Bases: *AxesMixin*, Boolean

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

Bases: *AxesMixin*, IntCategory

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

Bases: *AxesMixin*, Integer

class hist.axis.**NamedAxesTuple**(*_AxesTuple__iterable: Iterable[Axis]*)

Bases: *AxesTuple*

property label: tuple[str]

The labels of the axes. Defaults to name if label not given, or Axis N if neither was given.

property name: tuple[str]

The names of the axes. May be empty strings.

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

Bases: *AxesMixin*, Regular

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

Bases: *AxesMixin*, StrCategory

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

Bases: *AxesMixin*, Variable

Submodules

hist.axis.transform module

class hist.axis.transform.**AxisTransform**

Bases: object

forward(*value: float*) → float

Compute the forward transform

inverse(*value: float*) → float

Compute the inverse transform

class hist.axis.transform.**Function**(*forward: Any*, *inverse: Any*, *, *convert: Any | None = None*, *name: str = ''*)

Bases: *AxisTransform*

class hist.axis.transform.**Pow**(*power: float*)

Bases: *AxisTransform*

property power: float

The power of the transform

hist.dask package

class hist.dask.**Hist**(**in_args:* AxisProtocol | *Tuple[int, float, float]* | Storage | *str*, *storage:* Storage | *str* | *None = None*, *metadata: Any | None = None*, *data: ndarray[Any, dtype[Any]] | None = None*, *label: str | None = None*, *name: str | None = None*)

Bases: *Hist*, Histogram

class hist.dask.**NamedHist**(**args: Any*, ***kwargs: Any*)

Bases: *NamedHist*, Histogram

Submodules

hist.dask.hist module

class hist.dask.hist.**Hist**(**in_args: AxisProtocol | Tuple[int, float, float] | Storage | str, storage: Storage | str | None = None, metadata: Any | None = None, data: ndarray[Any, dtype[Any]] | None = None, label: str | None = None, name: str | None = None*)

Bases: *Hist*, Histogram

hist.dask.namedhist module

class hist.dask.namedhist.**NamedHist**(**args: Any, **kwargs: Any*)

Bases: *NamedHist*, Histogram

Submodules

hist.accumulators module

class hist.accumulators.**Mean**

Bases: pybind11_object

property count

fill(*self: boost_histogram._core.accumulators.Mean, value: object, *, weight: object = None*) → boost_histogram._core.accumulators.Mean

Fill the accumulator with values. Optional weight parameter.

property value

property variance

class hist.accumulators.**Sum**

Bases: pybind11_object

fill(*self: boost_histogram._core.accumulators.Sum, value: object*) → boost_histogram._core.accumulators.Sum

Run over an array with the accumulator

property value

class hist.accumulators.**WeightedMean**

Bases: pybind11_object

fill(*self: boost_histogram._core.accumulators.WeightedMean, value: object, *, weight: object = None*) → boost_histogram._core.accumulators.WeightedMean

Fill the accumulator with values. Optional weight parameter.

property sum_of_weights

property sum_of_weights_squared

property value

property variance

class hist.accumulators.WeightedSum

Bases: pybind11_object

fill(*self: boost_histogram._core.accumulators.WeightedSum*, *value: object*, *, *variance: object = None*) → boost_histogram._core.accumulators.WeightedSum

Fill the accumulator with values. Optional variance parameter.

property value

property variance

hist.axestuple module

class hist.axestuple.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 hist.axestuple.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 hist.axestuple.NamedAxesTuple(*_AxesTuple__iterable: Iterable[Axis]*)

Bases: *AxesTuple*

property label: tuple[str]

The labels of the axes. Defaults to name if label not given, or Axis N if neither was given.

property name: tuple[str]

The names of the axes. May be empty strings.

hist.basehist module

class hist.basehist.**BaseHist**(**in_args: AxisTypes | Storage | str, storage: Storage | str | None = None, metadata: Any = None, data: np.typing.NDArray[Any] | None = None, label: str | None = None, name: str | None = None*)

Bases: `Histogram`

property T: Self

density() → np.typing.NDArray[Any]

Density NumPy array.

fill(**args: Any, weight: Any | None = None, sample: Any | None = None, threads: int | None = None, **kwargs: Any*) → Self

Insert data into the histogram using names and indices, return a Hist object.

fill_flattened(**args: Any, weight: Any | None = None, sample: Any | None = None, threads: int | None = None, **kwargs: Any*) → Self

classmethod from_columns(*data: Mapping[str, Any], axes: Sequence[str | AxisProtocol], *, weight: str | None = None, storage: Storage = Double()*) → Self

integrate(*name: int | str, i_or_list: InnerIndexing | list[str | int] | None = None, j: InnerIndexing | None = None*) → Self | float | bh.accumulators.Accumulator

plot(**args: Any, overlay: str | None = None, **kwargs: Any*) → Hist1DArtists | Hist2DArtists

Plot method for BaseHist object.

plot1d(**, ax: matplotlib.axes.Axes | None = None, overlay: str | int | None = None, **kwargs: Any*) → Hist1DArtists

Plot1d method for BaseHist object.

plot2d(**, ax: matplotlib.axes.Axes | None = None, **kwargs: Any*) → Hist2DArtists

Plot2d method for BaseHist object.

plot2d_full(**, ax_dict: dict[str, matplotlib.axes.Axes] | None = None, **kwargs: Any*) → tuple[Hist2DArtists, Hist1DArtists, Hist1DArtists]

Plot2d_full method for BaseHist object.

Pass a dict of axes to `ax_dict`, otherwise, the current figure will be used.

plot_pie(**, ax: matplotlib.axes.Axes | None = None, **kwargs: Any*) → Any

plot_pull(*func: Callable[[np.typing.NDArray[Any]], np.typing.NDArray[Any]] | str, *, ax_dict: dict[str, matplotlib.axes.Axes] | None = None, **kwargs: Any*) → tuple[FitResultArtists, RatiolikeArtists]

`plot_pull` method for `BaseHist` object.

Return a tuple of artists following a structure of `(main_ax_artists, subplot_ax_artists)`

plot_ratio(*other: hist.BaseHist | Callable[[np.typing.NDArray[Any]], np.typing.NDArray[Any]] | str, *, ax_dict: dict[str, matplotlib.axes.Axes] | None = None, **kwargs: Any*) → tuple[MainAxisArtists, RatiolikeArtists]

`plot_ratio` method for `BaseHist` object.

Return a tuple of artists following a structure of `(main_ax_artists, subplot_ax_artists)`

profile(*axis: int | str*) → Self

Returns a profile (Mean/WeightedMean) histogram from a normal histogram with N-1 axes. The axis given is profiled over and removed from the final histogram.

project(**args: int | str*) → Self | float | Any

Projection of axis idx.

show(***kwargs: Any*) → Any

Pretty print histograms to the console.

sort(*axis: int | str*, *key: Callable[[int],* SupportsLessThan*] | Callable[[str],* SupportsLessThan*] | None = None*, *reverse: bool = False*) → Self

Sort a categorical axis.

stack(*axis: int | str*) → *Stack*

Returns a stack from a normal histogram axes.

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

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

class hist.basehist.**SupportsLessThan**(**args*, ***kwargs*)

Bases: Protocol

hist.basehist.**process_mistaken_quick_construct**(*axes: Sequence[*AxisProtocol | *Tuple[int, float, float]* | ConstructProxy*]*) → Generator[*AxisProtocol* | Tuple[int, float, float], None, None]

hist.classichist module

hist.classichist.**main**() → None

hist.hist module

class hist.hist.**Hist**(**in_args:* AxisProtocol | *Tuple[int, float, float]* | Storage | *str*, *storage:* Storage | *str* | *None = None*, *metadata: Any | None = None*, *data: ndarray[Any, dtype[Any]] | None = None*, *label: str | None = None*, *name: str | None = None*)

Bases: *BaseHist*

hist.intervals module

hist.intervals.**clopper_pearson_interval**(*num: ndarray[Any, dtype[Any]]*, *denom: ndarray[Any, dtype[Any]]*, *coverage: float | None = None*) → ndarray[Any, dtype[Any]]

Compute the Clopper-Pearson coverage interval for a binomial distribution. c.f. http://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval

Parameters

- **num** – Numerator or number of successes.
- **denom** – Denominator or number of trials.
- **coverage** – Central coverage interval. Default is one standard deviation, which is roughly 0.68.

Returns
The Clopper-Pearson central coverage interval.

hist.intervals.**poisson_interval**(*values: ndarray[Any, dtype[Any]]*, *variances: ndarray[Any, dtype[Any]] | None = None*, *coverage: float | None = None*) → ndarray[Any, dtype[Any]]

The Frequentist coverage interval for Poisson-distributed observations.

What is calculated is the "Garwood" interval, c.f. V. Patil, H. Kulkarni (Revstat, 2012) or http://ms.mcmaster.ca/peter/s743/poissonalpha.html. If `variances` is supplied, the data is assumed to be weighted, and the unweighted count is approximated by `values**2/variances`, which effectively scales the unweighted Poisson interval by the average weight. This may not be the optimal solution: see 10.1016/j.nima.2014.02.021 (arXiv:1309.1287) for a proper treatment.

In cases where the value is zero, an upper limit is well-defined only in the case of unweighted data, so if `variances` is supplied, the upper limit for a zero value will be set to `NaN`.

Parameters

- **values** – Sum of weights.
- **variances** – Sum of weights squared.
- **coverage** – Central coverage interval. Default is one standard deviation, which is roughly `0.68`.

Returns
The Poisson central coverage interval.

hist.intervals.**ratio_uncertainty**(*num: ndarray[Any, dtype[Any]]*, *denom: ndarray[Any, dtype[Any]]*, *uncertainty_type: Literal['poisson', 'poisson-ratio', 'efficiency'] = 'poisson'*) → Any

Calculate the uncertainties for the values of the ratio `num/denom` using the specified coverage interval approach.

Parameters

- **num** – Numerator or number of successes.
- **denom** – Denominator or number of trials.
- **uncertainty_type** – Coverage interval type to use in the calculation of the uncertainties.
  - `"poisson"` (default) implements the Garwood confidence interval for a Poisson-distributed numerator scaled by the denominator. See *hist.intervals.poisson_interval()* for further details.
  - `"poisson-ratio"` implements a confidence interval for the ratio `num / denom` assuming it is an estimator of the ratio of the expected rates from two independent Poisson distributions. It over-covers to a similar degree as the Clopper-Pearson interval does for the Binomial efficiency parameter estimate.
  - `"efficiency"` implements the Clopper-Pearson confidence interval for the ratio `num / denom` assuming it is an estimator of a Binomial efficiency parameter. This is only valid if the entries contributing to `num` are a strict subset of those contributing to `denom`.

Returns
The uncertainties for the ratio.

hist.namedhist module

class hist.namedhist.**NamedHist**(**args: Any*, ***kwargs: Any*)

Bases: *BaseHist*

fill(*weight: Any | None = None*, *sample: Any | None = None*, *threads: int | None = None*, ***kwargs: Any*) → Self

Insert data into the histogram using names and return a NamedHist object. NamedHist could only be filled by names.

fill_flattened(*obj: Any | None = None*, ***, *weight: Any | None = None*, *sample: Any | None = None*, *threads: int | None = None*, ***kwargs: Any*) → Self

project(**args: int | str*) → Self | float | Any

Projection of axis idx.

hist.numpy module

hist.numpy.**histogram**(*a: object*, *bins: int | str | ndarray[Any, dtype[Any]] = 10*, *range: tuple[float, float] | None = None*, *normed: None = None*, *weights: object | None = None*, *density: bool = False*, ***, *histogram: None | type[boost_histogram.Histogram] = None*, *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.

  New 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
```

New 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()
```

hist.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[boost_histogram.Histogram] = None*, *storage:* 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
```

(continues on next page)

(continued from previous page)

```
>>> 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()
```

hist.numpy.**histogramdd**(*a: tuple[object, ...]*, *bins: int | tuple[int, ...] | tuple[numpy.ndarray[Any, numpy.dtype[Any]], ...] = 10*, *range: None | Sequence[None | tuple[float, float]] = None*, *normed: None = None*, *weights: object | None = None*, *density: bool = False*, ***, *histogram: None | type[boost_histogram.Histogram] = None*, *storage:* 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, ...)*. 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)
```

hist.plot module

hist.plot.**hist2dplot**(*H*, *xbins=None*, *ybins=None*, *labels=None*, *cbar=True*, *cbarsize='7%'*, *cbarpad=0.2*, *cbarpos='right'*, *cbarextend=False*, *cmin=None*, *cmax=None*, *ax=None*, *flow='hint'*, ***kwargs*)

Create a 2D histogram plot from *np.histogram*-like inputs.

Parameters

- **H** (*object*) – Histogram object with containing values and optionally bins. Can be:
  - *np.histogram* tuple
  - *boost_histogram* histogram object
  - raw histogram values as list of list or 2d-array
- **xbins** (*1D array-like, optional, default None*) – Histogram bins along x axis, if not part of H.
- **ybins** (*1D array-like, optional, default None*) – Histogram bins along y axis, if not part of H.
- **labels** (*2D array (H-like) or bool, default None, optional*) – Array of per-bin labels to display. If True will display numerical values
- **cbar** (*bool, optional, default True*) – Draw a colorbar. In contrast to mpl behaviors the cbar axes is appended in such a way that it doesn't modify the original axes width:height ratio.
- **cbarsize** (*str or float, optional, default "7%"*) – Colorbar width.
- **cbarpad** (*float, optional, default 0.2*) – Colorbar distance from main axis.
- **cbarpos** (*{'right', 'left', 'bottom', 'top'}, optional, default "right"*) – Colorbar position w.r.t main axis.
- **cbarextend** (*bool, optional, default False*) – Extends figure size to keep original axes size same as without cbar. Only safe for 1 axes per fig.
- **cmin** (*float, optional*) – Colorbar minimum.
- **cmax** (*float, optional*) – Colorbar maximum.
- **ax** (*matplotlib.axes.Axes, optional*) – Axes object (if None, last one is fetched or one is created)
- **flow** (*str, optional {"show", "sum","hint", None}*) – Whether plot the under/overflow bin. If "show", add additional under/overflow bin. If "sum", add the under/overflow bin content to first/last bin. "hint" would highlight the bins with under/overflow contents
- ****kwargs** – Keyword arguments passed to underlying matplotlib function - pcolormesh.

Return type
Hist2DArtist

hist.plot.**histplot**(*H*, *bins=None*, ***, *yerr: ArrayLike | bool | None = None*, *w2=None*, *w2method=None*, *stack=False*, *density=False*, *binwnorm=None*, *histtype='step'*, *xerr=False*, *label=None*, *sort=None*, *edges=True*, *binticks=False*, *ax=None*, *flow='hint'*, ***kwargs*)

Create a 1D histogram plot from *np.histogram*-like inputs.

Parameters

- **H** (*object*) – Histogram object with containing values and optionally bins. Can be:
  - *np.histogram* tuple
  - PlottableProtocol histogram object
  - *boost_histogram* classic (<0.13) histogram object
  - raw histogram values, provided *bins* is specified.

  Or list thereof.
- **bins** (*iterable, optional*) – Histogram bins, if not part of `h`.
- **yerr** (*iterable or bool, optional*) – Histogram uncertainties. Following modes are supported: - True, sqrt(N) errors or poissonian interval when `w2` is specified - shape(N) array of for one sided errors or list thereof - shape(Nx2) array of for two sided errors or list thereof
- **w2** (*iterable, optional*) – Sum of the histogram weights squared for poissonian interval error calculation
- **w2method** (*callable, optional*) – Function calculating CLs with signature `low, high = fcn(w, w2)`. Here `low` and `high` are given in absolute terms, not relative to w. Default is `None`. If w2 has integer values (likely to be data) poisson interval is calculated, otherwise the resulting error is symmetric `sqrt(w2)`. Specifying `poisson` or `sqrt` will force that behaviours.
- **stack** (*bool, optional*) – Whether to stack or overlay non-axis dimension (if it exists). N.B. in contrast to ROOT, stacking is performed in a single call aka `histplot([h1, h2, ...], stack=True)` as opposed to multiple calls.
- **density** (*bool, optional*) – If true, convert sum weights to probability density (i.e. integrates to 1 over domain of axis) (Note: this option conflicts with `binwnorm`)
- **binwnorm** (*float, optional*) –

  If true, convert sum weights to bin-width-normalized, with unit equal to
  supplied value (usually you want to specify 1.)
- **histtype** (*{'step', 'fill', 'errorbar'}, optional, default: "step"*) – Type of histogram to plot:
  - "step": skyline/step/outline of a histogram using plt.step
  - "fill": filled histogram using plt.fill_between
  - "errorbar": single marker histogram using plt.errorbar
- **xerr** (*bool or float, optional*) – Size of xerr if `histtype == 'errorbar'`. If `True`, bin-width will be used.
- **label** (*str or list, optional*) – Label for legend entry.
- **sort** (*{'label'/'l', 'yield'/'y'}, optional*) – Append '_r' for reverse.
- **edges** (*bool, default: True, optional*) – Specifies whether to draw first and last edges of the histogram
- **binticks** (*bool, default: False, optional*) – Attempts to draw x-axis ticks coinciding with bin boundaries if feasible.
- **ax** (*matplotlib.axes.Axes, optional*) – Axes object (if None, last one is fetched or one is created)

- **flow** (*str, optional { "show", "sum", "hint", "none"}*) – Whether plot the under/overflow bin. If "show", add additional under/overflow bin. If "sum", add the under/overflow bin content to first/last bin.
- ****kwargs** – Keyword arguments passed to underlying matplotlib functions - {'step', 'fill_between', 'errorbar'}.

Return type
List[Hist1DArtists]

hist.plot.**plot2d_full**(*self:* BaseHist, *, *ax_dict: dict[str, matplotlib.axes._axes.Axes] | None = None*, ***kwargs: Any*) → tuple[mplhep.plot.ColormeshArtists, Union[mplhep.plot.StairsArtists, mplhep.plot.ErrorBarArtists], Union[mplhep.plot.StairsArtists, mplhep.plot.ErrorBarArtists]]

Plot2d_full method for BaseHist object.

Pass a dict of axes to `ax_dict`, otherwise, the current figure will be used.

hist.plot.**plot_pie**(*self:* BaseHist, *, *ax: Axes | None = None*, ***kwargs: Any*) → Any

hist.plot.**plot_pull_array**(*__hist:* BaseHist, *pulls: ndarray[Any, dtype[Any]]*, *ax: Axes*, *bar_kwargs: dict[str, Any]*, *pp_kwargs: dict[str, Any]*) → PullArtists

Plot a pull plot on the given axes

hist.plot.**plot_ratio_array**(*__hist:* BaseHist, *ratio: ndarray[Any, dtype[Any]]*, *ratio_uncert: ndarray[Any, dtype[Any]]*, *ax: Axes*, ***kwargs: Any*) → RatioErrorbarArtists | RatioBarArtists

Plot a ratio plot on the given axes

hist.plot.**plot_stack**(*self:* Stack, *, *ax: Axes | None = None*, *legend: bool | None = False*, ***kwargs: Any*) → Any

hist.quick_construct module

class hist.quick_construct.**ConstructProxy**(*hist_class: type[*BaseHist*]*, **axes:* AxisProtocol)

Bases: *QuickConstruct*

AtomicInt64(*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*) → *BaseHist*

Double(*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*) → *BaseHist*

Int64(*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*) → *BaseHist*

Mean(*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*) → *BaseHist*

Unlimited(*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*) → *BaseHist*

Weight(*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*) → *BaseHist*

WeightedMean(*, *metadata: Any = None*, *data: np.typing.NDArray[Any] | None = None*, *label: str | None = None*, *name: str | None = None*) → *BaseHist*

class hist.quick_construct.**MetaConstructor**

Bases: type

property new: *QuickConstruct*

class hist.quick_construct.**QuickConstruct**(*hist_class: type[BaseHist], *axes: AxisProtocol*)

Bases: object

Create a quick construct instance. This is the "base" quick constructor; it will always require at least one axes to be added before allowing a storage or fill to be performed.

Bool(*name: str = '', label: str = '', metadata: Any | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Boolean(*name: str = '', label: str = '', metadata: Any | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Func(*bins: int, start: float, stop: float, *, name: str = '', label: str = '', forward: Callable[[float], float], inverse: Callable[[float], float], metadata: Any | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Int(*start: int, stop: int, *, name: str = '', label: str = '', metadata: Any | None = None, flow: bool = True, underflow: bool | None = None, overflow: bool | None = None, growth: bool = False, circular: bool = False, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

IntCat(*categories: Iterable[int], *, name: str = '', label: str = '', metadata: Any | None = None, growth: bool = False, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

IntCategory(*categories: Iterable[int], *, name: str = '', label: str = '', metadata: Any | None = None, growth: bool = False, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Integer(*start: int, stop: int, *, name: str = '', label: str = '', metadata: Any | None = None, flow: bool = True, underflow: bool | None = None, overflow: bool | None = None, growth: bool = False, circular: bool = False, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Log(*bins: int, start: float, stop: float, *, name: str = '', label: str = '', metadata: Any | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Pow(*bins: int, start: float, stop: float, *, name: str = '', label: str = '', power: float, metadata: Any | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Reg(*bins: int, start: float, stop: float, *, name: str = '', label: str = '', metadata: Any | None = None, flow: bool = True, underflow: bool | None = None, overflow: bool | None = None, growth: bool = False, circular: bool = False, transform: AxisTransform | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Regular(*bins: int, start: float, stop: float, *, name: str = '', label: str = '', metadata: Any | None = None, flow: bool = True, underflow: bool | None = None, overflow: bool | None = None, growth: bool = False, circular: bool = False, transform: AxisTransform | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Sqrt(*bins: int, start: float, stop: float, *, name: str = '', label: str = '', metadata: Any | None = None, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

StrCat(*categories: Iterable[str], *, name: str = '', label: str = '', metadata: Any | None = None, growth: bool = False, __dict__: dict[str, Any] | None = None*) → *ConstructProxy*

StrCategory(*categories: Iterable[str]*, *, *name: str = ''*, *label: str = ''*, *metadata: Any | None = None*, *growth: bool = False*, *__dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Var(*edges: Iterable[float]*, *, *name: str = ''*, *label: str = ''*, *metadata: Any | None = None*, *flow: bool = True*, *underflow: bool | None = None*, *overflow: bool | None = None*, *growth: bool = False*, *circular: bool = False*, *__dict__: dict[str, Any] | None = None*) → *ConstructProxy*

Variable(*edges: Iterable[float]*, *, *name: str = ''*, *label: str = ''*, *metadata: Any | None = None*, *flow: bool = True*, *underflow: bool | None = None*, *overflow: bool | None = None*, *growth: bool = False*, *circular: bool = False*, *__dict__: dict[str, Any] | None = None*) → *ConstructProxy*

axes

hist_class

hist.stack module

class hist.stack.**Stack**(**args:* BaseHist)

Bases: object

property axes: *NamedAxesTuple*

classmethod from_dict(*d: Mapping[str,* BaseHist*]*) → Self

Create a Stack from a dictionary of histograms. The keys of the dictionary are used as names.

classmethod from_iter(*iterable: Iterable[*BaseHist*]*) → Self

Create a Stack from an iterable of histograms.

plot(*, *ax: mpl.axes.Axes | None = None*, ***kwargs: Any*) → Any

Plot method for Stack object.

project(**args: int | str*) → Self

Project the Stack onto a new axes.

show(***kwargs: object*) → Any

Pretty print the stacked histograms to the console.

hist.storage module

class hist.storage.**AtomicInt64**

Bases: atomic_int64, *Storage*

accumulator

alias of int

class hist.storage.**Double**

Bases: double, *Storage*

accumulator

alias of float

class hist.storage.**Int64**

Bases: int64, *Storage*

accumulator

alias of int

class hist.storage.**Mean**

Bases: mean, *Storage*

accumulator

alias of *Mean*

class hist.storage.**Storage**

Bases: object

accumulator: ClassVar[type[int] | type[float] | type[*boost_histogram.accumulators.WeightedMean*] | type[*boost_histogram.accumulators.WeightedSum*] | type[*boost_histogram.accumulators.Mean*]]

class hist.storage.**Unlimited**

Bases: unlimited, *Storage*

accumulator

alias of float

class hist.storage.**Weight**

Bases: weight, *Storage*

accumulator

alias of *WeightedSum*

class hist.storage.**WeightedMean**

Bases: weighted_mean, *Storage*

accumulator

alias of *WeightedMean*

hist.svgplots module

hist.svgplots.**html_hist**(*h:* BaseHist, *function: Callable[[*BaseHist*],* svg*]*) → *html*

hist.svgplots.**make_ax_text**(*ax: Axis*, ***kwargs:* SupportsStr) → *text*

hist.svgplots.**make_text**(*txt: str | float*, ***kwargs:* SupportsStr) → *text*

hist.svgplots.**svg_hist_1d**(*h:* BaseHist) → *svg*

hist.svgplots.**svg_hist_1d_c**(*h:* BaseHist) → *svg*

hist.svgplots.**svg_hist_2d**(*h:* BaseHist) → *svg*

hist.svgutils module

class hist.svgutils.**SupportsStr**(**args*, ***kwargs*)

Bases: Protocol

class hist.svgutils.**XML**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: object

property name: str

property start: str

class hist.svgutils.**circle**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**div**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**ellipse**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**g**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**html**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**line**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**p**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**polygon**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**polyline**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

class hist.svgutils.**rect**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

classmethod pad(*x: float*, *y: float*, *scale_x: float*, *scale_y: float*, *height: float*, *left_edge: float*, *right_edge: float*, *pad_x: float = 0*, *pad_y: float = 0*, *opacity: float = 1*, *stroke_width: float = 2*) → Self

class hist.svgutils.**svg**(**args:* XML | *str*, ***kwargs: str*)

Bases: *XML*

class hist.svgutils.**text**(**contents:* XML | SupportsStr, ***kargs:* SupportsStr)

Bases: *XML*

hist.tag module

class hist.tag.**Locator**(*offset: int = 0*)

Bases: object

NAME = ''

offset

class hist.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 hist.tag.**at**(*value: int*)

Bases: object

value

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

Bases: *Locator*

value

class hist.tag.**rebin**(*value: int*)

Bases: object

factor

hist.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.

hist.version module

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X