uhi

Release 0.4.0

Henry Schreiner, Hans Dembinski, Jim Pivarski

Oct 17, 2023

CONTENTS:

UHI is a library that helps connect other Histogramming libraries. It is primarily indented to be a guide and static type check helper; you do not need an runtime dependency on UHI. It currently does so with the following components:

UHI Indexing, which describes a powerful indexing system for histograms, designed to extend standard Array indexing for Histogram operations.

UHI Indexing+ (referred to as UHI+ for short), which describes a set of extensions to the standard indexing that make it easier to use on the command line.

The PlottableProtocol, which describes the minimal and complete set of requirements for a source library to produce and a plotting library to consume to plot a histogram, including error bars.

CHAPTER

ONE

INDEXING

This is the design document for Unified Histogram Indexing (UHI). Much of the original plan is now implemented in boost-histogram. Other histogramming libraries can implement support for this as well, and the "tag" functors, like `sum` and `loc` can be used between libraries.

1.1 Syntax

The following examples assume you have imported `loc`, `rebin`, `underflow`, and `overflow` from boost-histogram or any other library that implements UHI.

1.1.1 Access:

```
v = h[b]          # Returns bin contents, indexed by bin number
v = h[loc(b)]     # Returns the bin containing the value
v = h[loc(b) + 1] # Returns the bin above the one containing the value
v = h[underflow]  # Underflow and overflow can be accessed with special tags
```

1.1.2 Slicing:

```
h == h[:]             # Slice over everything
h2 = h[a:b]           # Slice of histogram (includes flow bins)
h2 = h[:b]            # Leaving out endpoints is okay
h2 = h[loc(v):]       # Slices can be in data coordinates, too
h2 = h[::rebin(2)]    # Modification operations (rebin)
h2 = h[a:b:rebin(2)]  # Modifications can combine with slices
h2 = h[::sum]         # Projection operations # (name may change)
h2 = h[a:b:sum]       # Adding endpoints to projection operations
h2 = h[0:len:sum]     #   removes under or overflow from the calculation
h2 = h[v, a:b]        #   A single value v is like v:v+1:sum
h2 = h[a:b, ...]      # Ellipsis work just like normal numpy
```

1.1.3 Setting

```
# Single values
h[b] = v          # Returns bin contents, indexed by bin number
h[loc(b)] = v     # Returns the bin containing the value
h[underflow] = v # Underflow and overflow can be accessed with special tags

h[...] = array(...) # Setting with an array or histogram sets the contents if the sizes␣
↪match
                    # Overflow can optionally be included if endpoints are left out
                    # The number of dimensions for non-scalars should match␣
↪(broadcasting works normally otherwise)
```

All of this generalizes to multiple dimensions. `loc(v)` could return categorical bins, but slicing on categories would (currently) not be allowed. These all return histograms, so flow bins are always preserved - the one exception is projection; since this removes an axis, the only use for the slice edges is to be explicit on what part you are interested for the projection. So an explicit (non-empty) slice here will case the relevant flow bin to be excluded.

`loc`, `project`, and `rebin` all live inside the histogramming package (like boost-histogram), but are completely general and can be created by a user using an explicit API (below). `underflow` and `overflow` also follow a general API.

One drawback of the syntax listed above is that it is hard to select an action to run on an axis or a few axes out of many. For this use case, you can pass a dictionary to the index, and that has the syntax `{axis:action}`. The actions are slices, and follow the rules listed above. This looks like:

```
h[{0: slice(None, None, bh.rebin(2))}] # rebin axis 0 by two
h[{1: slice(0, bh.loc(3.5))}]          # slice axis 1 from 0 to the data coordinate 3.5
h[{7: slice(0, 2, bh.rebin(4))}]       # slice and rebin axis 7
```

If you don't like manually building slices, you can use the `Slicer()` utility to recover the original slicing syntax inside the dict:

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

h[{0: s[::rebin(2)]}]   # rebin axis 0 by two
h[{1: s[0:loc(3.5)]}]   # slice axis 1 from 0 to the data coordinate 3.5
h[{7: s[0:2:rebin(4)]}] # slice and rebin axis 7
```

1.1.4 Invalid syntax:

```
h[1.0] # Floats are not allowed, just like numpy
h[::2] # Skipping is not (currently) supported
h[..., None] # None == np.newaxis is not supported
```

1.1.5 Reordering axes

It is not possible to reorder axis with this syntax; libraries are expected to provide a `.project(*axis: int)` method which provides a way to reorder, as well as fast access to a small subset of a large histogram in a complementary way to the above indexing.

1.1.6 Rejected proposals or proposals for future consideration, maybe `hist`-only:

```
h2 = h[1.0j:2.5j + 1] # Adding a j suffix to a number could be used in place of ``loc(x)``
h2 = h[1.0] # Floats in place of ``loc(x)``: too easy to make a mistake
```

1.2 Examples

For a histogram, the slice should be thought of like this:

```
histogram[start:stop:action]
```

The start and stop can be either a bin number (following Python rules), or a callable; the callable will get the axis being acted on and should return an extended bin number (`-1` and `len(ax)` are flow bins). A provided callable is `bh.loc`, which converts from axis data coordinates into bin number.

The final argument, `action`, is special. A general API is being worked on, but for now, `bh.sum` will "project out" or "integrate over" an axes, and `bh.rebin(n)` will rebin by an integral factor. Both work correctly with limits; `bh.sum` will remove flow bins if given a range. `h[0:len:bh.sum]` will sum without the flow bins.

Here are a few examples that highlight the functionality of UHI:

1.2.1 Example 1:

You want to slice axis 0 from 0 to 20, axis 1 from .5 to 1.5 in data coordinates, axis 2 needs to have double size bins (rebin by 2), and axis 3 should be summed over. You have a 4D histogram.

Solution:

```
ans = h[:20, bh.loc(-.5):bh.loc(1.5), ::bh.rebin(2), ::bh.sum]
```

1.2.2 Example 2:

You want to set all bins above 4.0 in data coordinates to 0 on a 1D histogram.

Solution:

```
h[bh.loc(4.0):] = 0
```

You can set with an array, as well. The array can either be the same length as the range you give, or the same length as the range + under/overflows if the range is open ended (no limit given). For example:

```
h = bh.Histogram(bh.axis.Regular(10, 0, 1))
h[:] = np.ones(10) # underflow/overflow still 0
h[:] = np.ones(12) # underflow/overflow now set too
```

Note that for clarity, while basic NumPy broadcasting is supported, axis-adding broadcasting is not supported; you must set a 2D histogram with a 2D array or a scalar, not a 1D array.

1.2.3 Example 3:

You want to sum from -infinity to 2.4 in data coordinates in axis 1, leaving all other axes alone. You have an ND histogram, with N >= 2.

Solution:

```
ans = h[:, :bh.loc(2.4):bh.sum, ...]
```

Notice that last example could be hard to write if the axis number, 1 in this case, was large or programmatically defined. In these cases, you can pass a dictionary of `{axis:slice}` into the indexing operation. A shortcut to quickly generate slices is provided, as well:

```
ans = h[{1: slice(None,bh.loc(2.4),bh.sum)}]

# Identical:
s = bh.tag.Slicer()
ans = h[{1: s[:bh.loc(2.4):bh.sum]}]
```

1.2.4 Example 4:

You want the underflow bin of a 1D histogram.

Solution:

```
val = h1[bh.underflow]
```

1.3 Details

1.3.1 Implementation notes

loc, rebin, and sum are *not* unique tags, or special types, but rather APIs for classes. New versions of these could be added, and implementations could be shared among Histogram libraries. For clarity, the following code is written in Python 3.6+. Prototype here. Extra doc here.

Note that the API comes in two forms; the `__call__`/`__new__` operator form is more powerful, slower, optional, and is currently not supported by boost-histogram. A fully conforming UHI implementation must allow the tag form without the operators.

Basic implementation example (WIP):

```
class loc:
    "When used in the start or stop of a Histogram's slice, x is taken to be the␣
↪position in data coordinates."
    def __init__(self, value, offset):
        self.value = value
        self.offset = offset
```

(continues on next page)

(continued from previous page)

```
    # supporting __add__ and __sub__ also recommended

    def __call__(self, axis):
        return axis.index(self.value) + self.offset

# Other flags, such as callable functions, could be added and detected later.

# UHI will perform a maximum performance sum when python's sum is encountered

def underflow(axis):
    return -1
def overflow(axis):
    return len(axis)


class rebin:
    """
    When used in the step of a Histogram's slice, rebin(n) combines bins,
    scaling their widths by a factor of n. If the number of bins is not
    divisible by n, the remainder is added to the overflow bin.
    """
    def __init__(self, factor):
        # Items with .factor are specially treated in boost-histogram,
        # performing a high performance rebinning
        self.factor = factor

    # Optional and not used by boost-histogram
    def __call__(self, binning, axis, counts):
        factor = self.factor
        if isinstance(binning, Regular):
            indexes = (numpy.arange(0, binning.num, factor),)

            num, remainder = divmod(binning.num, factor)
            high, hasover = binning.high, binning.hasover

            if binning.hasunder:
                indexes[0][:] += 1
                indexes = ([0],) + indexes

            if remainder == 0:
                if binning.hasover:
                    indexes = indexes + ([binning.num + int(binning.hasunder)],)
            else:
                high = binning.left(indexes[-1][-1])
                hasover = True

            binning = Regular(num, binning.low, high, hasunder=binning.hasunder,␣
↪hasover=hasover)
            counts = numpy.add.reduceat(counts, numpy.concatenate(indexes), axis=axis)
            return binning, counts
```

(continues on next page)

(continued from previous page)

```
        else:
            raise NotImplementedError(type(binning))
```

CHAPTER

TWO

INDEXING+

This is an extended version of UHI, called UHI+. This is not implemented in boost-histogram, but is implemented in Hist.

2.1 Syntax extensions

UHI+ avoids using the standard tags found in UHI by using more advanced Python syntax.

2.1.1 Location based slicing/access: numeric axes

You can replace location based indexing `loc(1.23)` → `1.23j` (a "j" suffix on a number literal). You can shift by an integer, just like with loc: `2.3j + 1` will be one bin past the one containing the location "2.3".

```
v = h[2j]     # Returns the bin containing "2.0"
v = h[2j + 1] # Returns the bin above the one containing "2.0"
h2 = h[2j:]   # Slices starting with the bin containing "2.0"
```

2.1.2 Location based slicing/access: string axis

If you have a string based axis, you can use a string directly `loc("label")` → `"label"`.

```
v = h["a"]    # Returns the "a" bin (string category axis)
```

2.1.3 Rebinning

You can replace `rebin(2)` → `2j` in the third slot of a slice.

```
h2 = h[::2j]    # Modification operations (rebin)
h2 = h[a:b:2j]  # Modifications can combine with slices
```

2.1.4 Named based indexing

An optional extension to indexing is expected for histogram implementations that support names. If named axes are supported, any expression that refers to an axis by an integer can also refer to it by a name string. `.project(*axis: int | str)` is probably the most common place to see this, but you can also use strings in the UHI dict access, such as:

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

h[{"a": s[::2j]}]       # rebin axis "a" by two
h[{"x": s[0:3.5j]}]     # slice axis "x" from 0 to the data coordinate 3.5
h[{"other": s[0:2:4j]}] # slice and rebin axis "other"
```

CHAPTER

THREE

PLOTTING

This is a description of the `PlottableProtocol`. Any plotting library that accepts an object that follows the `PlottableProtocol` can plot object that follow this protocol, and libraries that follow this protocol are compatible with plotters. The Protocol is runtime checkable, though as usual, that will only check for the presence of the needed methods at runtime, not for the static types.

3.1 Using the protocol:

Plotters should only depend on the methods and attributes listed below. In short, they are:

- `h.kind`: The `bh.Kind` of the histogram (COUNT or MEAN)
- `h.values()`: The value (as given by the kind)
- `h.variances()`: The variance in the value (None if an unweighed histogram was filled with weights)
- `h.counts()`: How many fills the bin received or the effective number of fills if the histogram is weighted
- `h.axes`: A Sequence of axes

Axes have:

- `ax[i]`: A tuple of (lower, upper) bin, or the discrete bin value (integer or string)
- `len(ax)`: The number of bins
- Iteration is supported
- `ax.traits.circular`: True if circular
- `ax.traits.discrete`: True if the bin represents a single value (e.g. Integer or Category axes) instead of an interval (e.g. Regular or Variable axes)

Plotters should see if `.counts()` is None; no boost-histogram objects currently return None, but a future storage or different library could.

Also check `.variances`; if not None, this storage holds variance information and error bars should be included. Boost-histogram histograms will return something unless they know that this is an invalid assumption (a weighted fill was made on an unweighted histogram).

To statically restrict yourself to valid API usage, use `PlottableHistogram` as the parameter type to your function (Not needed at runtime).

3.2 Implementing the protocol:

Add UHI to your MyPy environment; an example `.pre-commit-config.yaml` file:

```
- repo: https://github.com/pre-commit/mirrors-mypy
  rev: v0.812
  hooks:
  - id: mypy
    files: src
    additional_dependencies: [uhi, numpy~=1.20.1]
```

Then, check your library against the Protocol like this:

```
from typing import TYPE_CHECKING, cast

if TYPE_CHECKING:
    _: PlottableHistogram = cast(MyHistogram, None)
```

3.3 Help for plotters

The module `uhi.numpy_plottable` has a utility to simplify the common use case of accepting a PlottableProtocol or other common formats, primarily a NumPy `histogram/histogram2d/histogramdd` tuple. The `ensure_plottable_histogram` function will take a histogram or NumPy tuple, or an object that implements `.to_numpy()` or `.numpy()` and convert it to a `NumPyPlottableHistogram`, which is a minimal implementation of the Protocol. By calling this function on your input, you can then write your plotting function knowing that you always have a `PlottableProtocol` object, greatly simplifying your code.

3.4 The full protocol version 1.2 follows:

(Also available as `uhi.typing.plottable.PlottableProtocol`, for use in tests, etc.

```
"""
Using the protocol:

Producers: use isinstance(myhist, PlottableHistogram) in your tests; part of
the protocol is checkable at runtime, though ideally you should use MyPy; if
your histogram class supports PlottableHistogram, this will pass.

Consumers: Make your functions accept the PlottableHistogram static type, and
MyPy will force you to only use items in the Protocol.
"""

from __future__ import annotations

import sys
from collections.abc import Iterator, Sequence
from typing import Any, Tuple, TypeVar, Union

# NumPy 1.20+ will work much, much better than previous versions when type checking
```

(continues on next page)

(continued from previous page)

```
import numpy as np

if sys.version_info < (3, 8):
    from typing_extensions import Protocol, runtime_checkable

else:
    from typing import Protocol, runtime_checkable


protocol_version = (1, 2)

# Known kinds of histograms. A Producer can add Kinds not defined here; a
# Consumer should check for known types if it matters. A simple plotter could
# just use .value and .variance if non-None and ignore .kind.
#
# Could have been Kind = Literal["COUNT", "MEAN"] - left as a generic string so
# it can be extendable.
Kind = str

# Implementations are highly encouraged to use the following construct:
# class Kind(str, enum.Enum):
#     COUNT = "COUNT"
#     MEAN = "MEAN"
# Then return and use Kind.COUNT or Kind.MEAN.


@runtime_checkable
class PlottableTraits(Protocol):
    @property
    def circular(self) -> bool:
        """
        True if the axis "wraps around"
        """

    @property
    def discrete(self) -> bool:
        """
        True if each bin is discrete - Integer, Boolean, or Category, for example
        """


T_co = TypeVar("T_co", covariant=True)


@runtime_checkable
class PlottableAxisGeneric(Protocol[T_co]):
    # name: str - Optional, not part of Protocol
    # label: str - Optional, not part of Protocol
    #
    # Plotters are encouraged to plot label if it exists and is not None, and
    # name otherwise if it exists and is not None, but these properties are not
    # available on all histograms and not part of the Protocol.
```

(continues on next page)

(continued from previous page)

```
    @property
    def traits(self) -> PlottableTraits:
        ...

    def __getitem__(self, index: int) -> T_co:
        """
        Get the pair of edges (not discrete) or bin label (discrete).
        """

    def __len__(self) -> int:
        """
        Return the number of bins (not counting flow bins, which are ignored
        for this Protocol currently).
        """

    def __eq__(self, other: Any) -> bool:
        """
        Required to be sequence-like.
        """

    def __iter__(self) -> Iterator[T_co]:
        """
        Useful element of a Sequence to include.
        """


PlottableAxisContinuous = PlottableAxisGeneric[Tuple[float, float]]
PlottableAxisInt = PlottableAxisGeneric[int]
PlottableAxisStr = PlottableAxisGeneric[str]

PlottableAxis = Union[PlottableAxisContinuous, PlottableAxisInt, PlottableAxisStr]


@runtime_checkable
class PlottableHistogram(Protocol):
    @property
    def axes(self) -> Sequence[PlottableAxis]:
        ...

    @property
    def kind(self) -> Kind:
        ...

    # All methods can have a flow=False argument - not part of this Protocol.
    # If this is included, it should return an array with flow bins added,
    # normal ordering.

    def values(self) -> np.typing.NDArray[Any]:
        """
        Returns the accumulated values. The counts for simple histograms, the
        sum of weights for weighted histograms, the mean for profiles, etc.
```

(continues on next page)

(continued from previous page)

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

    def variances(self) -> np.typing.NDArray[Any] | None:
        """
        Returns the estimated variance of the accumulated values. The sum of squared
        weights for weighted histograms, the variance of samples for profiles, etc.
        For an unweighed histogram where kind == "COUNT", this should return the same
        as values if the histogram was not filled with weights, and None otherwise.

        If counts is equal to 1 or less, the variance in that cell is undefined if
        kind == "MEAN".

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

    def counts(self) -> np.typing.NDArray[Any] | None:
        """
        Returns the number of entries in each bin for an unweighted
        histogram or profile and an effective number of entries (defined below)
        for a weighted histogram or profile. An exotic generalized histogram could
        have no sensible .counts, so this is Optional and should be checked by
        Consumers.

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

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

        A suggested implementation is:

            return np.divide(
                sum_of_weights**2,
                sum_of_weights_squared,
                out=np.zeros_like(sum_of_weights, dtype=np.float64),
                where=sum_of_weights_squared != 0)
        """
```

CHAPTER

FOUR

CHANGELOG

4.1 v0.4.0: Version 0.4.0

Released on 2023-10-17 - GitHub - PyPI

4.2 v0.3.3: Version 0.3.3

Released on 2023-01-04 - GitHub - PyPI

4.3 v0.3.2: Version 0.3.2

Released on 2022-09-20 - GitHub - PyPI

4.4 v0.3.1: Version 0.3.1

Released on 2022-01-06 - GitHub - PyPI

4.5 v0.3.0: Version 0.3.0

Released on 2021-06-15 - GitHub - PyPI

4.6 v0.2.1: Version 0.2.1

Released on 2021-03-18 - GitHub - PyPI

4.7 v0.2.0: Version 0.2.0

Released on 2021-03-17 - GitHub - PyPI

4.8 v0.1.2: Version 0.1.2

Released on 2021-03-09 - GitHub - PyPI

4.9 v0.1.1: Version 0.1.1

Released on 2021-01-29 - GitHub - PyPI

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search