boost-histogram and hist

Henry Schreiner

June 19, 2019
Histograms in Python
Current state of histograms in Python

Core library: numpy
- Historically slow
- No histogram object
- Plotting is separate

Other libraries
- Narrow focus: speed, plotting, or language
- Many are abandoned
- Poor design, backends, distribution
### What is needed?

#### Design
- A histogram should be an object
- Manipulation and plotting should be easy

#### Performance
- Fast single threaded filling
- Multithreaded filling (since it’s 2019)

#### Flexibility
- Axes options: sparse, growing, labels
- Storage: integers, weights, errors...

#### Distribution
- Easy to use anywhere, pip or conda
- Should have wheels, be easy to build, etc.
Future of histograms in Python

Core histogramming libraries

- boost-histogram
- ROOT

Universal adaptor

Aghast

Front ends (plotting, etc)

- hist
- mpl-hep
- physt
- others

Henry Schreiner

June 19, 2019
Boost.Histogram (C++14)
Intro to Boost.Histogram

- Multidimensional templated header-only histogram library: [.github.com/boostorg/histogram](https://github.com/boostorg/histogram)
- Designed by Hans Dembinski, inspired by ROOT, GSL, and histbook

### Histogram

- Axes
- Storages
- Accumulators

### Axes types

- Regular, Circular
- Variable
- Integer
- Category

### Storage

- Static
- Dynamic

- Regular axis with log transform
- Optional underflow
- Optional overflow
- Accumulator: int, double, unlimited, ...

---

Boost 1.70 now released with Boost.Histogram!

June 19, 2019
Intro to Boost.Histogram

- Multidimensional templated header-only histogram library: [boostorg/histogram](https://boostorg/histogram)
- Designed by Hans Dembinski, inspired by ROOT, GSL, and histbook

### Histogram
- Axes
- Storages
- Accumulators

### Axes types
- Regular, Circular
- Variable
- Integer
- Category

---

**Boost 1.70 now released with Boost.Histogram!**
boost-histogram (Python)
Intro to the Python bindings

• Boost.Histogram developed with Python in mind
• Original bindings based on Boost::Python
  ▶ Hard to build and distribute
  ▶ Somewhat limited
• New bindings: /scikit-hep/boost-histogram
  ▶ 0-dependency build (C++14 only)
  ▶ State-of-the-art PyBind11
Design

- 260+ unit tests run on Azure on Linux, macOS, and Windows
- Up to 16 axes supported (may go up or down)

Resembles the original Boost.Histogram where possible, with changes where needed for Python performance and idioms.

C++
#include <boost/histogram.hpp>
namespace bh = boost::histogram;

auto hist = bh::make_histogram(
    bh::axis::regular<>{2, 0, 1, "x"},
    bh::axis::regular<>{4, 0, 1, "y"});

hist(.2, .3);
hist(.4, .5);
hist(.3, .2);

Python
import boost.histogram as bh

hist = bh.histogram(
    bh.axis.regular(2, 0, 1, metadata="x"),
    bh.axis.regular(4, 0, 1, metadata="y"))

hist.fill(
    [.2, .4, .3],
    [.3, .5, .2])

Henry Schreiner
**Design: Manipulations**

- **Combine** two histograms
  \[ \text{hist1} + \text{hist2} \]

- **Scale** a histogram
  \[ \text{hist} \times 2.0 \]

- **Sum** a histogram contents
  \[ \text{hist}.\text{sum}() \]

- **Access** an axis
  \[ \text{axis0} = \text{hist}.\text{axis}(0) \]
  \[ \text{axis0}.\text{edges}() \quad \# \text{The edges array} \]
  \[ \text{axis0}.\text{bin}(1) \quad \# \text{The bin accessors} \]

- **Fill** 2D histogram with values or arrays
  \[ \text{hist}.\text{fill}(x, y) \]

- **Fill** copies in 4 threads, then merge
  \[ \text{hist}.\text{fill}(x, y, \text{threads}=4) \]

- **Fill** in 4 threads (atomic storage only)
  \[ \text{hist}.\text{fill}(x, y, \text{atomic}=4) \]

- **Convert** to Numpy, 0-copy
  \[ \text{hist}.\text{view}() \]
Planned: Histogram slicing syntax

See #35.

Access:

\[ v = h[b] \]  
# Returns bin contents, indexed by bin number

\[ v = h[\text{loc}(b)] \]  
# Returns the bin containing the value

Setting

\[ h[...] = \text{np.ndarray}(...) \]  
# Setting with an array or histogram sets the contents if the sizes match
Slicing:

\[ h == h[: \] \]  \quad \# \text{Slice over everything}

\[ h2 = h[a:b] \]  \quad \# \text{Slice of histogram (includes flow bins)}

\[ h2 = h[:b] \]  \quad \# \text{Leaving out endpoints is okay}

\[ h2 = h[loc(v):] \]  \quad \# \text{Slices can be in data coordinates, too}

\[ h2 = h[::\text{project}] \]  \quad \# \text{Projection operations}

\[ h2 = h[::\text{rebin}(2)] \]  \quad \# \text{Modification operations (rebin)}

\[ h2 = h[a:b:\text{rebin}(2)] \]  \quad \# \text{Modifications can combine with slices}

\[ h2 = h[a:b:\text{project}] \]  \quad \# \text{Adding endpoints to projection operation removes}
\quad \# \text{under or overflow from the calculation}

\[ h2 = h[0:\text{len}(h2.\text{axis}(0)):\text{project}] \]  \quad \# \text{Projection without flow bins}

\[ h2 = [a:b, \ldots] \]  \quad \# \text{Ellipsis work just like normal numpy}

\[ h2 = h[::\text{rebin}(\text{loc}(\text{width}))] \]  \quad \# \text{WIP: This should be considered}
Planned: Histogram slicing syntax (3)

Invalid syntax:

```python
h[v, a:b]  # You cannot mix slices and bin contents access (h is an integer)
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
```
Flexibility: Axis

- `bh.axis.regular`
  - `bh.axis.regular_uoflow`
  - `bh.axis.regular_noflow`
  - `bh.axis.regular_growth`
- `bh.axis.circular`
- `bh.axis.regular_log`
- `bh.axis.regular_sqrt`
- `bh.axis.regular_pow`
- `bh.axis.integer`
- `bh.axis.integer_noflow`
- `bh.axis.integer_growth`
- `bh.axis.variable`
- `bh.axis.category_int`
- `bh.axis.category_int_growth`
Flexibility: Storage types

- bh.storage.int
- bh.storage.double
- bh.storage.unlimited (WIP)
- bh.storage.atomic_int
- bh.storage.weight (WIP)
- bh.storage.profile (WIP, needs sampled fill)
- bh.storage.weighted_profile (WIP, needs sampled fill)
The following classic measurements are with:

**1D**
- 100 regular bins
- 10,000,000 entries

**2D**
- 100x100 regular bins
- 1,000,000 entries

See my [histogram performance post](#) for measurements of other libraries.
## Performance: macOS, dual core, 1D

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage</th>
<th>Fill</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy</td>
<td>uint64</td>
<td></td>
<td>149.4 ms</td>
<td>1x</td>
</tr>
<tr>
<td>Any</td>
<td>int</td>
<td></td>
<td>236 ms</td>
<td>0.63x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td></td>
<td>86.23 ms</td>
<td>1.7x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 1</td>
<td></td>
<td>132 ms</td>
<td>1.1x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 2</td>
<td></td>
<td>168.2 ms</td>
<td>0.89x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 4</td>
<td></td>
<td>143.6 ms</td>
<td>1x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 1</td>
<td></td>
<td>84.75 ms</td>
<td>1.8x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 2</td>
<td></td>
<td>51.6 ms</td>
<td>2.9x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 4</td>
<td></td>
<td>42.39 ms</td>
<td>3.5x</td>
</tr>
<tr>
<td>Type</td>
<td>Storage</td>
<td>Fill</td>
<td>Time</td>
<td>Speedup</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>Numpy</td>
<td>uint64</td>
<td></td>
<td>121 ms</td>
<td>1x</td>
</tr>
<tr>
<td>Any</td>
<td>int</td>
<td></td>
<td>261.5 ms</td>
<td>0.46x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td></td>
<td>142.2 ms</td>
<td>0.85x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>1</td>
<td>319.1 ms</td>
<td>0.38x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>48</td>
<td>272.9 ms</td>
<td>0.44x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>1</td>
<td>243.4 ms</td>
<td>0.5x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>6</td>
<td>94.76 ms</td>
<td>1.3x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>12</td>
<td>71.38 ms</td>
<td>1.7x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>24</td>
<td>52.26 ms</td>
<td>2.3x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>48</td>
<td>43.01 ms</td>
<td>2.8x</td>
</tr>
</tbody>
</table>
### Performance: KNL, 64 core, 1D (anaconda)

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage</th>
<th>Fill</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy</td>
<td>uint64</td>
<td></td>
<td>716.9 ms</td>
<td>1x</td>
</tr>
<tr>
<td>Any</td>
<td>int</td>
<td></td>
<td>1418 ms</td>
<td>0.51x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td></td>
<td>824 ms</td>
<td>0.87x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>1</td>
<td>871.7 ms</td>
<td>0.82x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>4</td>
<td>437.1 ms</td>
<td>1.6x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>64</td>
<td>198.8 ms</td>
<td>3.6x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>128</td>
<td>186.8 ms</td>
<td>3.8x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>256</td>
<td>195.2 ms</td>
<td>3.7x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>1</td>
<td>796.9 ms</td>
<td>0.9x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>2</td>
<td>430.6 ms</td>
<td>1.7x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>4</td>
<td>247.6 ms</td>
<td>2.9x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>64</td>
<td>88.77 ms</td>
<td>8.1x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>128</td>
<td>98.08 ms</td>
<td>7.3x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>256</td>
<td>112.2 ms</td>
<td>6.4x</td>
</tr>
</tbody>
</table>
### Performance: macOS, dual core, 2D

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage</th>
<th>Fill</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy</td>
<td>uint64</td>
<td></td>
<td>121.1 ms</td>
<td>1x</td>
</tr>
<tr>
<td>Any</td>
<td>int</td>
<td></td>
<td>37.12 ms</td>
<td>3.3x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td></td>
<td>18.5 ms</td>
<td>6.5x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>1</td>
<td>20.21 ms</td>
<td>6x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>2</td>
<td>14.17 ms</td>
<td>8.5x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint</td>
<td>4</td>
<td>10.23 ms</td>
<td>12x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>1</td>
<td>17.86 ms</td>
<td>6.8x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>2</td>
<td>9.41 ms</td>
<td>13x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td>4</td>
<td>6.854 ms</td>
<td>18x</td>
</tr>
</tbody>
</table>
### Performance: CentOS7, 24 core, 2D (anaconda)

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage</th>
<th>Fill</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy</td>
<td>uint64</td>
<td></td>
<td>87.27 ms</td>
<td>1x</td>
</tr>
<tr>
<td>Any</td>
<td>int</td>
<td></td>
<td>41.42 ms</td>
<td>2.1x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td></td>
<td>21.67 ms</td>
<td>4x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 1</td>
<td></td>
<td>38.61 ms</td>
<td>2.3x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 6</td>
<td></td>
<td>19.89 ms</td>
<td>4.4x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 24</td>
<td></td>
<td>9.556 ms</td>
<td>9.1x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 48</td>
<td></td>
<td>8.518 ms</td>
<td>10x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 1</td>
<td></td>
<td>36.5 ms</td>
<td>2.4x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 6</td>
<td></td>
<td>8.976 ms</td>
<td>9.7x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 12</td>
<td></td>
<td>5.318 ms</td>
<td>16x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 24</td>
<td></td>
<td>4.388 ms</td>
<td>20x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 48</td>
<td></td>
<td>5.839 ms</td>
<td>15x</td>
</tr>
</tbody>
</table>

Henry Schreiner

boost-histogram and hist

June 19, 2019
## Performance: KNL, 64 core, 2D (anaconda)

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage</th>
<th>Fill</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy</td>
<td>uint64</td>
<td></td>
<td>439.5 ms</td>
<td>1x</td>
</tr>
<tr>
<td>Any</td>
<td>int</td>
<td></td>
<td>250.6 ms</td>
<td>1.8x</td>
</tr>
<tr>
<td>Regular</td>
<td>int</td>
<td></td>
<td>135.6 ms</td>
<td>3.2x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 1</td>
<td></td>
<td>142.2 ms</td>
<td>3.1x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 4</td>
<td></td>
<td>52.71 ms</td>
<td>8.3x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 32</td>
<td></td>
<td>12.05 ms</td>
<td>36x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 64</td>
<td></td>
<td>16.5 ms</td>
<td>27x</td>
</tr>
<tr>
<td>Regular</td>
<td>aint 256</td>
<td></td>
<td>43.93 ms</td>
<td>10x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 1</td>
<td></td>
<td>141.1 ms</td>
<td>3.1x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 2</td>
<td></td>
<td>70.78 ms</td>
<td>6.2x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 4</td>
<td></td>
<td>36.11 ms</td>
<td>12x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 64</td>
<td></td>
<td>18.93 ms</td>
<td>23x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 128</td>
<td></td>
<td>36.09 ms</td>
<td>12x</td>
</tr>
<tr>
<td>Regular</td>
<td>int 256</td>
<td></td>
<td>55.64 ms</td>
<td>7.9x</td>
</tr>
</tbody>
</table>
## Performance: Summary

<table>
<thead>
<tr>
<th>System</th>
<th>1D max speedup</th>
<th>2D max speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>macOS 1 core</td>
<td>1.7 x</td>
<td>6.5 x</td>
</tr>
<tr>
<td>macOS 2 core</td>
<td>3.5 x</td>
<td>18 x</td>
</tr>
<tr>
<td>Linux 1 core</td>
<td>0.85 x</td>
<td>4 x</td>
</tr>
<tr>
<td>Linux 24 core</td>
<td>2.8 x</td>
<td>20 x</td>
</tr>
<tr>
<td>KNL 1 core</td>
<td>0.87 x</td>
<td>3.2 x</td>
</tr>
<tr>
<td>KNL 64 core</td>
<td>8.1 x</td>
<td>36 x</td>
</tr>
</tbody>
</table>

- Note that Numpy 1D is well optimized (last few versions)
- Anaconda versions may provide a few more optimizations to Numpy
- Mixing axes types in boost-histogram can reduce performance by 2-3x
We must provide excellent distribution.
  ▶ If anyone writes `pip install boost-histogram` and it fails, we have failed.

- Docker ManyLinux1 GCC 9: /scikit-hep/manylinuxgcc
- Used in /scikit-hep/iMinuit, see /scikit-hep/azure-wheel-helpers

**Wheels**
- manylinux1 32 and 64 bit, Py 2.7 & 3.5+
- manylinux2010 64 bit, Py 2.7 & 3.5+
- macOS 10.9+ 64 bit, Py 2.7 & 3.6+
- Windows 32 and 64 bit, Py 2.7 & 3.6+

**Source**
- SDist
- Build directly from GitHub

**Conda**
- conda-forge package planned when released

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

- Add `from_numpy` and numpy style shortcut(s)
- Filling improvements (speed, flexibility)
- Support edges/centers matrices and direct setting
- Add reduce operations
- Release to PyPI
- Add more docs
Let's discuss API! (On GitHub issues or gitter)

- Download: pip install boost-histogram (WIP)
- Use: import boost.histogram as bh
- Create: hist = bh.histogram(bh.axis.regular(12,0,1))
- Fill: hist.fill(values)
- Access values, convert to numpy, etc.

Documentation
- The documentation will also need useful examples, feel free to contribute!
See #18
hist
hist is the ‘wrapper’ piece that does plotting and interacts with the rest of the ecosystem.

**Plans**
- Easy plotting adaptors (mpl-hep)
- Serialization formats (ROOT, HDF5)
- Auto-multithreading
- Statistical functions (Like TEfficiency)
- Multihistograms (HistBook)
- Interaction with fitters (ZFit, GooFit, etc)
- Bayesian Blocks algorithm from SciKit-HEP
- Command line histograms for stream of numbers

**Call for contributions**
- What do you need?
- What do you want?
- What would you like?

Join in the development! This should combine the best features of other packages.
• Supported by IRIS-HEP, NSF OAC-1836650