Histogram interoperability

Jim Pivarski

Princeton University – DIANA-HEP

October 1, 2018
## Histogramming in Python

<table>
<thead>
<tr>
<th>pip?</th>
<th>name</th>
<th>last release</th>
<th>interface style</th>
<th>depends on</th>
<th>integrates with</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PyROOT</td>
<td>2018</td>
<td>HEP</td>
<td>ROOT</td>
<td>numpy</td>
</tr>
<tr>
<td></td>
<td>YODA</td>
<td>2018</td>
<td>HEP</td>
<td>compiled</td>
<td>matplotlib, yaml</td>
</tr>
<tr>
<td>✓</td>
<td>physt</td>
<td>2018</td>
<td>HEP + data science</td>
<td>numpy</td>
<td>pandas, xarray, dask, protobuf, matplotlib, vega (plotting), folium (maps)</td>
</tr>
<tr>
<td>✓</td>
<td>fast-histogram</td>
<td>2018</td>
<td>simple (astronomy)</td>
<td>numpy</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>qhist</td>
<td>2018</td>
<td>HEP</td>
<td>ROOT</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>rootpy</td>
<td>2017</td>
<td>HEP</td>
<td>ROOT</td>
<td>pytables, matplotlib, stats</td>
</tr>
<tr>
<td>✓</td>
<td>Vaex (vaex.io)</td>
<td>2017</td>
<td>all-in-one GUI for big data, fast heatmaps</td>
<td>many!</td>
<td>Jupyter, matplotlib, HDF5, pandas, C++</td>
</tr>
<tr>
<td>✓</td>
<td>hdrhistogram</td>
<td>2017</td>
<td>“high dynamic range”</td>
<td>compiled</td>
<td>Java, C++</td>
</tr>
<tr>
<td>✓</td>
<td>multihist</td>
<td>2017</td>
<td>numpy wrapper</td>
<td>numpy</td>
<td>matplotlib</td>
</tr>
<tr>
<td>✓</td>
<td>matplotlib-hep</td>
<td>2016</td>
<td>HEP</td>
<td>matplotliblib</td>
<td>numpy, scipy</td>
</tr>
<tr>
<td>✓</td>
<td>pyhistogram</td>
<td>2014</td>
<td>HEP</td>
<td>numpy</td>
<td>matplotliblib, datetime</td>
</tr>
<tr>
<td>✓</td>
<td>histogram</td>
<td>2011</td>
<td>HEP</td>
<td>numpy</td>
<td>matplotliblib, HDF5</td>
</tr>
<tr>
<td>✓</td>
<td>SimpleHist</td>
<td>2011</td>
<td>HEP</td>
<td>numpy, matplotlib</td>
<td>ROOT</td>
</tr>
<tr>
<td>✓</td>
<td>paida</td>
<td>2007</td>
<td>HEP</td>
<td></td>
<td>AIDA!</td>
</tr>
<tr>
<td></td>
<td>theodoregoetz</td>
<td>never</td>
<td>HEP</td>
<td></td>
<td>scipy, uncertainties</td>
</tr>
</tbody>
</table>
Over the years, I’ve written five

<table>
<thead>
<tr>
<th>pip?</th>
<th>name</th>
<th>last release</th>
<th>interface style</th>
<th>depends on</th>
<th>integrates with</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ploton</td>
<td>2006</td>
<td>HEP</td>
<td>ROOT</td>
<td>SVG</td>
</tr>
<tr>
<td></td>
<td>SVGFig</td>
<td>2008</td>
<td>HEP</td>
<td>none!</td>
<td>SVG</td>
</tr>
<tr>
<td></td>
<td>Cassius</td>
<td>2013</td>
<td>HEP + data science</td>
<td>numpy</td>
<td>SVG, Augustus (Open Data Group)</td>
</tr>
<tr>
<td>√</td>
<td>Histogrammar</td>
<td>2016</td>
<td>combinational library</td>
<td>numpy</td>
<td>Spark, Julia, CUDA, ROOT (Cling), matplotlib, Bokeh, Vega</td>
</tr>
<tr>
<td>√</td>
<td>histbook</td>
<td>2018</td>
<td>HEP</td>
<td>numpy</td>
<td>Spark, Pandas, Vega</td>
</tr>
</tbody>
</table>
Over the years, I’ve written five

<table>
<thead>
<tr>
<th>pip? name</th>
<th>last release</th>
<th>interface style</th>
<th>depends on</th>
<th>integrates with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plothon</td>
<td>2006</td>
<td>HEP</td>
<td>ROOT</td>
<td>SVG</td>
</tr>
<tr>
<td>SVGFig</td>
<td>2008</td>
<td>HEP</td>
<td><em>none!</em></td>
<td>SVG</td>
</tr>
<tr>
<td>Cassius</td>
<td>2013</td>
<td>HEP + data science</td>
<td>numpy</td>
<td>SVG, Augustus (Open Data Group)</td>
</tr>
<tr>
<td>√ Histogrammar</td>
<td>2016</td>
<td>combinational library</td>
<td>numpy</td>
<td>Spark, Julia, CUDA, ROOT (Cling), matplotlib, Bokeh, Vega</td>
</tr>
<tr>
<td>√ histbook</td>
<td>2018</td>
<td>HEP</td>
<td>numpy</td>
<td>Spark, Pandas, Vega</td>
</tr>
</tbody>
</table>

*Implementing* data analysis tools isn’t the problem, it’s designing the *right* tool, one that solves the Cuisinart Problem:

It slices and dices, but is it worth the setup and cleanup?

Experimentation in this area is valuable.
All these libraries are overwhelming!

However, the problem isn’t that data analysts are using different tools; the problem is communicating results between them.
There is a standard: ROOT files

Even if the user wants to work in a non-ROOT way or avoid ROOT dependencies, we still have to be able to share results using the ROOT file format.

Look at industry:

- dozens of big data SQL engines that all run on Parquet files;
- dozens of machine learning tools that all run on Numpy from HDF5 files.

The most conservative part of a software ecosystem is its persistence format.

ROOT files are incredibly well established in HEP. It would take a strong technical argument/corner case to use something else.
uproot version 3 has write support, but (currently) only for histograms.

```python
>>> import uproot
>>> import numpy
>>> f = uproot.recreate("tmp.root")
>>> f["name"] = numpy.histogram(numpy.random.normal(0, 1, 100000))
```

Read it back in ROOT:

```python
>>> import ROOT
>>> f = ROOT.TFile("tmp.root")
>>> h = f.Get("name")
>>> h.Draw()```

![Histogram](image.png)
Bidirectional conversions to and from many libraries

Save a Physt plot to a ROOT file:

```python
>>> import physt
>>> f = uproot.recreate("tmp.root")
>>> f["name"] = physt.h1(numpy.random.normal(0, 1, 100000), bins=16,
... range=(-4, 4), name="physt histogram")
```

Read it back as Physt:

```python
>>> f["name"].physt().plot()
```

Read it back as a Numpy contents-and-edges tuple:

```python
>>> f["name"].numpy()
(array([ 23, 112, 473, 1652, 4370, 9061, 15024, 19213, 19091, 15019, 9128, 4501, 1671, 512, 128, 16], dtype=int32),
array([-4. , -3.5, -3. , -2.5, -2. , -1.5, -1. , -0.5, 0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ]))
```

Read it in ROOT:

```python
>>> f = ROOT.TFile("tmp.root")
>>> h = f.Get("name")
>>> h.Draw()
```
Some “non-histograms:” HEPData archival format (YAML)

```python
>>> print(f[name].hepdata())

independent_variables:
  - header: {name: physt histogram, units: null}
    values:
    - {low: -4.0, high: -3.5}
    - {low: -3.5, high: -3.0}
    - {low: -3.0, high: -2.5}
    - {low: -2.5, high: -2.0}
    - {low: -2.0, high: -1.5}
    - {low: -1.5, high: -1.0}
    - {low: -1.0, high: -0.5}
    - {low: -0.5, high: 0.0}
    - {low: 0.0, high: 0.5}
    - {low: 0.5, high: 1.0}
    - {low: 1.0, high: 1.5}
    - {low: 1.5, high: 2.0}
    - {low: 2.0, high: 2.5}
    - {low: 2.5, high: 3.0}
    - {low: 3.0, high: 3.5}
    - {low: 3.5, high: 4.0}

dependent_variables:
  - header: {name: counts, units: null}
    qualifiers: []
    values:
    - value: 23.0
      errors:
      - {symerror: 4.795831523312719, label: stat}
    - value: 112.0
      errors:
      - {symerror: 10.583005244258363, label: stat}
    - value: 473.0
      errors:
      - {symerror: 21.748563170931547, label: stat}
    - value: 1652.0
      errors:
      - {symerror: 40.64480286580315, label: stat}
    - value: 4370.0
      errors:
      - {symerror: 66.10597552415364, label: stat}
    - value: 9061.0
      errors:
      - {symerror: 95.1892851112981, label: stat}
    - value: 15024.0
      errors:
      - {symerror: 122.5724275683565, label: stat}
    - value: 19213.0
      errors:
      - {symerror: 138.61096637712328, label: stat}
    - value: 19091.0
      errors:
      - {symerror: 138.17018491700733, label: stat}
    - value: 15019.0
      errors:
      - {symerror: 122.55202976695246, label: stat}
    - value: 9128.0
      errors:
      - {symerror: 95.5405672999695, label: stat}
    - value: 4501.0
      errors:
      - {symerror: 67.08949247087803, label: stat}
```

8 / 18
Some “non-histograms:” Pandas DataFrame

```python
>>> f["name"].pandas()

<table>
<thead>
<tr>
<th>physt histogram</th>
<th>count</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-inf, -4.0)</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(-4.0, -3.5)</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>(-3.5, -3.0)</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>(-3.0, -2.5)</td>
<td>473</td>
<td>473</td>
</tr>
<tr>
<td>(-2.5, -2.0)</td>
<td>1652</td>
<td>1652</td>
</tr>
<tr>
<td>(-2.0, -1.5)</td>
<td>4370</td>
<td>4370</td>
</tr>
<tr>
<td>(-1.5, -1.0)</td>
<td>9061</td>
<td>9061</td>
</tr>
<tr>
<td>(-1.0, -0.5)</td>
<td>15024</td>
<td>15024</td>
</tr>
<tr>
<td>(-0.5, 0.0)</td>
<td>19213</td>
<td>19213</td>
</tr>
<tr>
<td>[0.0, 0.5)</td>
<td>19091</td>
<td>19091</td>
</tr>
<tr>
<td>[0.5, 1.0)</td>
<td>15019</td>
<td>15019</td>
</tr>
<tr>
<td>[1.0, 1.5)</td>
<td>9128</td>
<td>9128</td>
</tr>
<tr>
<td>[1.5, 2.0)</td>
<td>4501</td>
<td>4501</td>
</tr>
<tr>
<td>[2.0, 2.5)</td>
<td>1671</td>
<td>1671</td>
</tr>
<tr>
<td>[2.5, 3.0)</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>[3.0, 3.5)</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>[3.5, 4.0)</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>[4.0, inf)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The index for this DataFrame is an IntervalIndex.

The variance column is the “sumw2” (identical to count when filled without weights).

There’s room here for profile plots to share the same binning.

See the F.A.S.T. project: Olivier Davignon, Lukasz Kreczko, Ben Krikler, Jacob Linacre, Emmanuel Olatunji Olaiya, & Tai Sakuma.
We can add more without upsetting uproot

The code that understands streamers and how to write a TH1 is in uproot.

The code that understands how to convert to and from other libraries is in uproot-methods. We can make changes to uproot-methods independently of (and more rapidly than) uproot.
What about Histogrammar and histbook?

histogrammar

 histbōōk
Redesigned histogramming as a combinational library: you put simple pieces together to build n-dimensional histograms and profiles.

**Histograms:**

```
Bin(num, low, high, fillRule, Count())
```

**Three-dimensional histograms:**

```
Bin(xnum, xlow, xhigh, xfill,
    Bin(ynum, ylow, yhigh, yfill,
        Bin(znum, zlow, zhigh, zfill, Count())))
```

**Profile plots:**

```
Bin(xnum, xlow, xhigh, xfill,
    Deviate(yfill))
```

where `Deviate` aggregates a mean and standard deviation.

**Mix and match binning methods:**

```
SparselyBin(0.01, filleta,
            Bin(314, -3.14, 3.14, fillphi,
                Count()))
```
However, it was too general. An arbitrary tree of aggregators produces an arbitrary tree of aggregated data, which is hard to format as a plot.

The cartesian grid of axis types and storage types in Boost-Histogram and ROOT 7 histograms (and histbook) is general enough.
histbook: the “hist” part

Make cartesian, n-dimensional histograms that you can assign to plot facets on the fly.

```python
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1\times q2 < 0"),
...   split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.step("mass")
```
histbook: the “hist” part

Make cartesian, n-dimensional histograms that you can assign to plot facets on the fly.

```python
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
...                  split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.overlay("q1*q2 < 0").step("mass")
```
histbook: the “hist” part

Make cartesian, n-dimensional histograms that you can assign to plot facets on the fly.

```python
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1\times q2 < 0"),
...    split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.stack("q1\times q2 < 0").area("mass")
```

![Histogram illustration](image)

```python
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1\times q2 < 0"),
...    split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.stack("q1\times q2 < 0").area("mass")
```
histbook: the “hist” part

Make cartesian, n-dimensional histograms that you can assign to plot facets on the fly.

```python
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
...                  split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.beside("q1*q2 < 0").step("mass")
```

![Histogram example](image-url)
Make cartesian, n-dimensional histograms that you can assign to plot facets on the fly.

```python
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
    ...      split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.below("mt1").beside("mt2").step("mass")
```
histbook: the “hist” part

Make cartesian, n-dimensional histograms that you can assign to plot facets on the fly.

```python
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
...     split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.below("mt1").beside("mt2").overlay("q1*q2 < 0").step("mass")
```

![Histograms](image.png)
Problem: physicists define semantic relationships between a large set (thousands) of histograms three or four times:

1. when they construct the histograms (defining the binning)
2. when they fill the histograms
3. when they fit with CMS Combine or ATLAS HistFactory
4. when they publish results on HEPData
Problem: physicists define semantic relationships between a large set (thousands) of histograms three or four times:

1. when they construct the histograms (defining the binning)
2. when they fill the histograms
3. when they fit with CMS Conbine or ATLAS HistFactory
4. when they publish results on HEPData

Solution to 1–2: use functional programming to put the “fill rule” next to the binning: Histogrammar, histbook, RDataFrame, and AlphaTwirl (F.A.S.T. ingestion) all do this.
Solution to 2–3–4?: declare trees of related histograms and communicate this relationship to fitters:

```python
everything = ChannelsBook(
    mass = SamplesBook(
        SystematicsBook(
            Hist(bin("x", 5, 0, 5), systematic=[0]),
            Hist(bin("x + epsilon", 5, 0, 5), systematic=[1]),
            Hist(bin("x - epsilon", 5, 0, 5), systematic=[-1]))),
    truth = SamplesBook(
        Book(par1=Hist(bin("par1", 5, 0, 5)),
            par2=Hist(bin("par2", 5, 0, 5))))
)
```

A Book has a fill method like Hist, so these collections of histograms can be filled in one pass and retain their internal relationships, to be reused for fitting.

This is research: what’s the right way to structure histograms for fitting that still makes sense for filling? Need to work with developers of fitting libraries.
Third idea: Pandas DataFrames should be histograms

A DataFrame with an IntervalIndex is a sparse histogram.

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th></th>
<th>two</th>
<th></th>
<th>three</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.0, 1.0)</td>
<td>4538</td>
<td></td>
<td>[1.0, 2.0)</td>
<td>71</td>
<td>[6.0, 7.0)</td>
<td>41</td>
</tr>
<tr>
<td>[1.0, 2.0)</td>
<td>4513</td>
<td></td>
<td>[2.0, 3.0)</td>
<td>5009</td>
<td>[7.0, 8.0)</td>
<td>4708</td>
</tr>
<tr>
<td>[2.0, 3.0)</td>
<td>483</td>
<td></td>
<td>[3.0, 4.0)</td>
<td>4868</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3.0, 4.0)</td>
<td>4</td>
<td></td>
<td>[4.0, 5.0)</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[4.0, 5.0)</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[6.0, 7.0)</td>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[7.0, 8.0)</td>
<td>4708</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Third idea: Pandas DataFrames should be histograms

A DataFrame with an IntervalIndex is a *sparse* histogram.

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
<th>three</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0.0, 1.0)</td>
<td>[1.0, 2.0)</td>
<td>[6.0, 7.0)</td>
</tr>
<tr>
<td></td>
<td>4538</td>
<td>71</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>[1.0, 2.0)</td>
<td>[2.0, 3.0)</td>
<td>[7.0, 8.0)</td>
</tr>
<tr>
<td></td>
<td>4513</td>
<td>5009</td>
<td>4708</td>
</tr>
<tr>
<td></td>
<td>[2.0, 3.0)</td>
<td>[3.0, 4.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>483</td>
<td>4868</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[3.0, 4.0)</td>
<td>[4.0, 5.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[4.0, 5.0)</td>
<td>[6.0, 7.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[6.0, 7.0)</td>
<td>[7.0, 8.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4708</td>
<td>4708</td>
<td></td>
</tr>
</tbody>
</table>

Adding and other operations would just “do the right thing” if not for frustrating details like imputing NaN when an interval key is missing, rather than zero.

```python
>>> def add(*args):
...     return reduce(lambda x, y: x.add(y, fill_value=0), args)
```

<table>
<thead>
<tr>
<th></th>
<th>add(one, two)</th>
<th>add(two, three)</th>
<th>add(one, two, three)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0.0, 1.0)</td>
<td>[1.0, 2.0)</td>
<td>[0.0, 1.0)</td>
</tr>
<tr>
<td></td>
<td>4538</td>
<td>71</td>
<td>4538</td>
</tr>
<tr>
<td></td>
<td>[1.0, 2.0)</td>
<td>[2.0, 3.0)</td>
<td>[1.0, 2.0)</td>
</tr>
<tr>
<td></td>
<td>4584</td>
<td>5009</td>
<td>4584</td>
</tr>
<tr>
<td></td>
<td>[2.0, 3.0)</td>
<td>[3.0, 4.0)</td>
<td>[2.0, 3.0)</td>
</tr>
<tr>
<td></td>
<td>5492</td>
<td>4868</td>
<td>5492</td>
</tr>
<tr>
<td></td>
<td>[3.0, 4.0)</td>
<td>[4.0, 5.0)</td>
<td>[3.0, 4.0)</td>
</tr>
<tr>
<td></td>
<td>4872</td>
<td>52</td>
<td>4872</td>
</tr>
<tr>
<td></td>
<td>[4.0, 5.0)</td>
<td>[6.0, 7.0)</td>
<td>[4.0, 5.0)</td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>41</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>[6.0, 7.0)</td>
<td>[7.0, 8.0)</td>
<td>[6.0, 7.0)</td>
</tr>
<tr>
<td></td>
<td>4708</td>
<td>4708</td>
<td>4708</td>
</tr>
</tbody>
</table>

Wrap it as a subclass?
Conclusions

- The design of data analysis tools like histogramming has an impact on human efficiency. It’s worth optimizing (experimentally!).

- The chaos of different tools can be mitigated by a common persistence format, and ROOT’s the obvious candidate.

- Some of the ideas in Histogrammar and histbook are also in Boost-Histogram, Physt, F.A.S.T., and ROOT 7. Some aren’t. I’d rather contribute ideas than maintain my own library.

- Looking forward to finding ways to collaborate, get feedback from users... and iterate!