Histogram interoperability

Jim Pivarski

Princeton University – DIANA-HEP

October 1, 2018

Histogramming in Python

pip?	name	last release	interface style	depends on	integrates with
	PyROOT	2018	HEP	ROOT	numpy
	YODA	2018	HEP	*compiled*	matplotlib, yaml
√	physt	2018	HEP + data science	numpy	pandas, xarray, dask, protobuf, matplotlib, vega (plotting), folium (maps)
√	fast-histogram	2018	simple (astronomy)	numpy	
√	qhist	2018	HEP	ROOT	
√	rootpy	2017	HEP	ROOT	pytables, matplotlib, stats
√	Vaex (vaex.io)	2017	all-in-one GUI for big data, fast heatmaps	*many!*	Jupyter, matplotlib, HDF5, pandas, C++
√	hdrhistogram	2017	"high dynamic range"	*compiled*	Java, C++
√	multihist	2017	numpy wrapper	numpy	matplotlib
√	matplotlib-hep	2016	HEP	matplotlib	numpy, scipy
√	pyhistogram	2014	HEP	numpy	matplotlib, datetime
√	histogram	2011	HEP	numpy	matplotlib, HDF5
√	SimpleHist	2011	HEP	numpy, matplotlib	ROOT
√	paida	2007	HEP		AIDA!
	theodoregoetz	never	HEP	scipy, uncertainties	numpy, matplotlib

Over the years, I've written five

pip?	name	last release	interface style	depends on	integrates with
	Plothon	2006	HEP	ROOT	SVG
	SVGFig	2008	HEP	*none!*	SVG
	Cassius	2013	HEP + data science	numpy	SVG, Augustus (Open Data Group)
$\surd$	Histogrammar	2016	combinational library	numpy	Spark, Julia, CUDA, ROOT (Cling), matplotlib, Bokeh, Vega
$\surd$	histbook	2018	HEP	numpy	Spark, Pandas, Vega

Over the years, I've written five

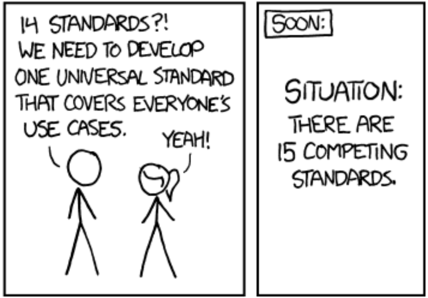
pip?	name	last release	interface style	depends on	integrates with
	Plothon	2006	HEP	ROOT	SVG
	SVGFig	2008	HEP	*none!*	SVG
	Cassius	2013	HEP + data science	numpy	SVG, Augustus (Open Data Group)
√	Histogrammar	2016	combinational library	numpy	Spark, Julia, CUDA, ROOT (Cling), matplotlib, Bokeh, Vega
√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

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.

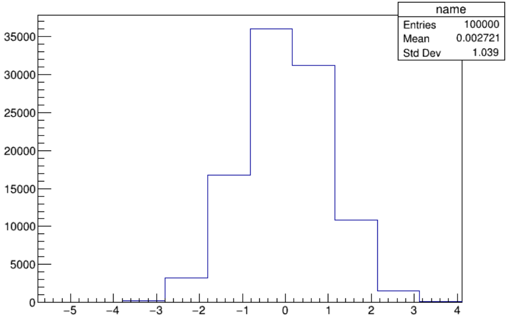
Writing histograms with uproot

uproot version 3 has write support, but (currently) only for histograms.

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

Read it back in ROOT:

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



Bidirectional conversions to and from many libraries

Save a Physt plot to a ROOT file:

```
>>> 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:

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

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

```
>>> f["name"].numpy()
```

```
(array([   23,   112,   473,  1652,  4370,  9061, 15024, 19213, 19091, 15019,  9128,  4501,  1671,   512,   128,    16]
 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:

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

Some "non-histograms:" HEPData archival format (YAML)

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


```yaml
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}
```

Some "non-histograms:" Pandas DataFrame

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

physt histogram	count	variance
[-inf, -4.0)	5	5
[-4.0, -3.5)	23	23
[-3.5, -3.0)	112	112
[-3.0, -2.5)	473	473
[-2.5, -2.0)	1652	1652
[-2.0, -1.5)	4370	4370
[-1.5, -1.0)	9061	9061
[-1.0, -0.5)	15024	15024
[-0.5, 0.0)	19213	19213
[0.0, 0.5)	19091	19091
[0.5, 1.0)	15019	15019
[1.0, 1.5)	9128	9128
[1.5, 2.0)	4501	4501
[2.0, 2.5)	1671	1671
[2.5, 3.0)	512	512
[3.0, 3.5)	128	128
[3.5, 4.0)	16	16
[4.0, inf)	1	1

The index for this DataFrame is an `IntervalIndex`.

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

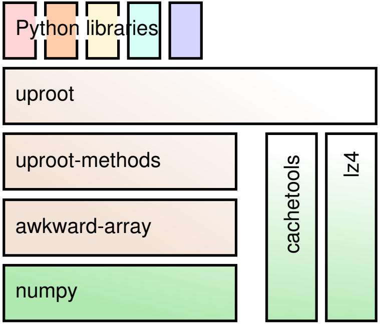
There's room here for profile plots to share the same binning.

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?

histo·*grammar*

/histōˌˈgɹæm.ər/

histbook

Histogrammar

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()))
```

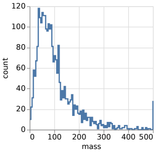
However, it was too general. An arbitrary tree of aggregators produces an arbitrary tree of aggregated data, which is hard to format as a plot.

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.

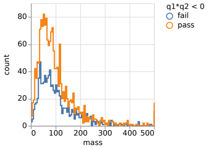
```
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
...                  split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.step("mass")
```



histbook: the "hist" part

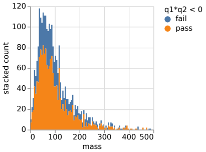
Make cartesian, n-dimensional histograms that you can assign to plot facets on the fly.

```
>>> 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")
```



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

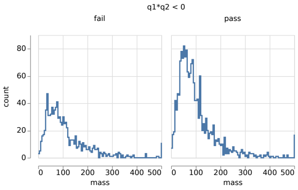
```
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
...                  split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.stack("q1*q2 < 0").area("mass")
```



histbook: the "hist" part

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

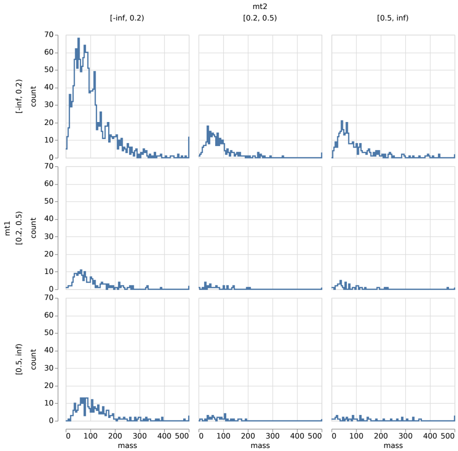
```
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
...                  split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.beside("q1*q2 < 0").step("mass")
```



histbook: the "hist" part

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

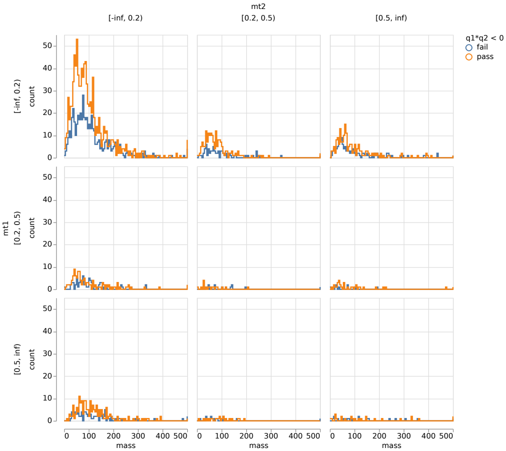
```
>>> from histbook import *
>>> multihist = Hist(bin("mass", 100, 0, 500), cut("q1*q2 < 0"),
...                  split("mt1", [0.2, 0.5]), split("mt2", [0.2, 0.5]), fill=df)
>>> multihist.below("mt1").beside("mt2").step("mass")
```



histbook: the "hist" part

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

```
>>> 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")
```



histbook: the "book" part

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

histbook: the “book” part

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.

histbook: the "book" part

Solution to 2–3–4?: declare trees of related histograms and communicate this relationship to fitters:

```
everything = ChannelsBook(
    mass = SamplesBook(["data", "signal", "background"],
                  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(["signal", "background"],
                        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.

```
                 one                      two                        three
    ----------------         ----------------          -----------------
    [0.0, 1.0)  4538         [1.0, 2.0)    71          [6.0, 7.0)      41
    [1.0, 2.0)  4513         [2.0, 3.0)  5009          [7.0, 8.0)    4708
    [2.0, 3.0)   483         [3.0, 4.0)  4868
    [3.0, 4.0)     4         [4.0, 5.0)    52
```

Third idea: Pandas DataFrames should be histograms

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

```
              one                    two                     three
-----------------      -----------------      ------------------
[0.0, 1.0)   4538      [1.0, 2.0)     71      [6.0, 7.0)      41
[1.0, 2.0)   4513      [2.0, 3.0)   5009      [7.0, 8.0)    4708
[2.0, 3.0)    483      [3.0, 4.0)   4868
[3.0, 4.0)      4      [4.0, 5.0)     52
```

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.

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

```
   add(one, two)         add(two, three)        add(one, two, three)
-----------------      -----------------      ---------------------
[0.0, 1.0)   4538      [1.0, 2.0)     71      [0.0, 1.0)       4538
[1.0, 2.0)   4584      [2.0, 3.0)   5009      [1.0, 2.0)       4584
[2.0, 3.0)   5492      [3.0, 4.0)   4868      [2.0, 3.0)       5492
[3.0, 4.0)   4872      [4.0, 5.0)     52      [3.0, 4.0)       4872
[4.0, 5.0)     52      [6.0, 7.0)     41      [4.0, 5.0)         52
                       [7.0, 8.0)   4708      [6.0, 7.0)         41
                                              [7.0, 8.0)       4708
```

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!