SuperhistogramS
— or —
abstract algebra for fun and profit

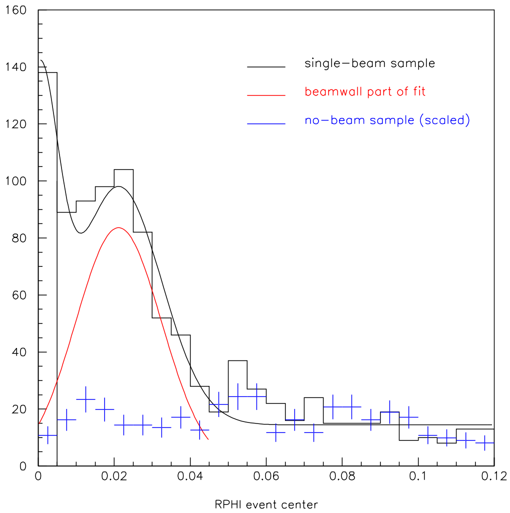
Jim Pivarski

Princeton University – IRIS-HEP

November 15, 2023

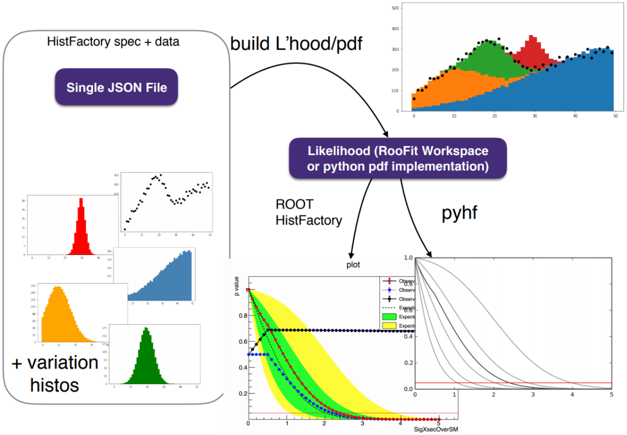
Vague statement of the problem

Long, long ago, histograms were individual objects that were managed individually.



Vague statement of the problem

Now, histograms are more often used collectively, with thousands of histograms in a single fit.



Vague statement of the problem

A "superhistogram" is a large collection of histograms that are meant to be interpreted together.

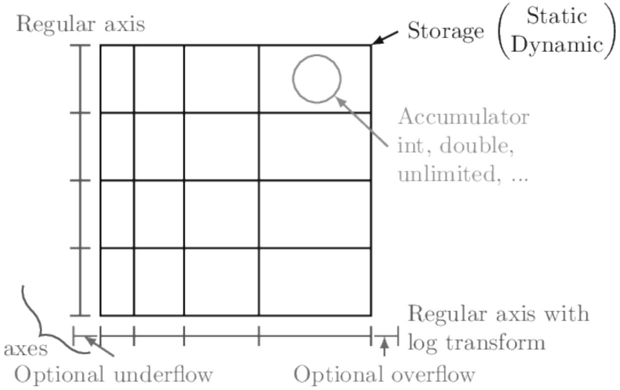
Vague statement of the problem

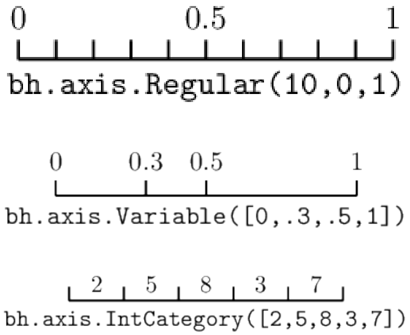
Representing a superhistogram with a directory of ordinary histograms is

- wasteful because much of the same metadata is copied in memory or on disk, and many small buffers of bin contents is less efficient than one big buffer.
- inconvenient because the object with a common meaning has to be managed as individual objects without an explicit connection. (Often in practice, they're only linked by naming conventions.)

Vague statement of the problem

Boost::Histogram provides a generic way to create an *n*-dimensional space with regular, variable, and categorical axes.





But a Boost::Histogram is still a single histogram:

- *n* axes form an *n*-dimensional space,
- each scalar `fill` operation increments *one* bin in that space.

Vague statement of the problem

But a Boost::Histogram is still a single histogram:

- *n* axes form an *n*-dimensional space,
- each scalar `fill` operation increments *one* bin in that space.

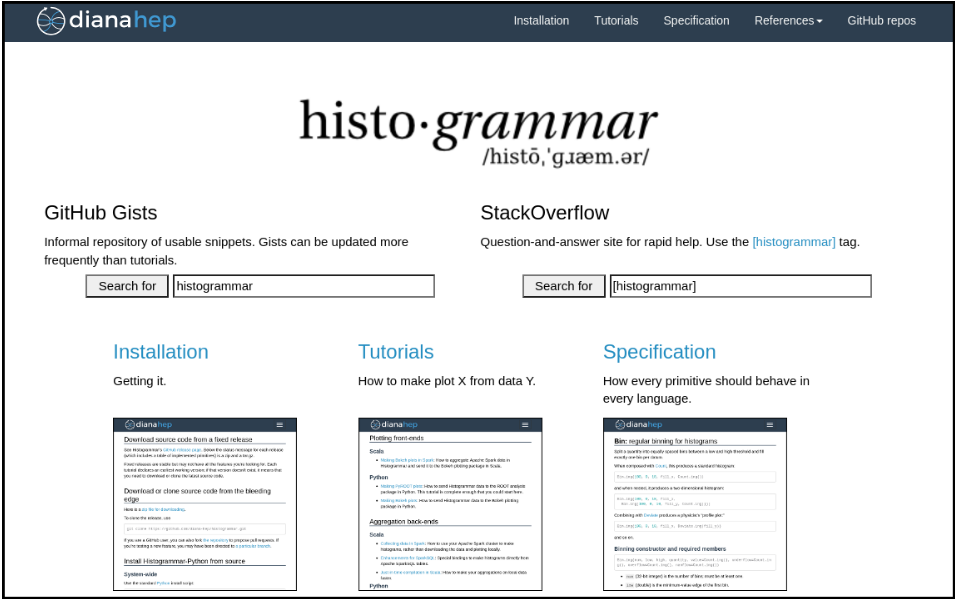
A superhistogram has multiple sources, channels, and systematics.

- Not all histograms in the collection have the same number of bins or the same dimensions, but many do.
- One `fill` of the superhistogram would increment every histogram in the collection.

Something like this

```
h = SuperHist(SuperHist(
        SuperHist(
            Hist.new.Reg(100, 0, 30, name="syst-up"),
            Hist.new.Reg(100, 0, 30, name="nominal"),
            Hist.new.Reg(100, 0, 30, name="syst-down"),
            name="pt",
        ),
        SuperHist(
            Hist.new.Reg(50, -5, 5, name="syst-up"),
            Hist.new.Reg(50, -5, 5, name="nominal"),
            Hist.new.Reg(50, -5, 5, name="syst-down"),
            name="eta",
        ),
        name="data",
    ),
    ...,        # similarly for name="mc"
)
h.fill(df)    # one DataFrame row is a scalar fill operation
```

So far, this is looking like Histogrammar



R.I.P. Histogrammar

Histogrammar was too loose: it described a *tree* of nested axes (unlike Boost::Histogram's *sequence*), and then there was no connection among the branches of that tree.

R.I.P. Histogrammar

Histogrammar was too loose: it described a *tree* of nested axes (unlike Boost::Histogram's *sequence*), and then there was no connection among the branches of that tree.

We need a relationship that restricts the superhistogram to a useful subset of all possible trees.

Abstract algebra for fun (profit comes later)

Many operations on data structures can be described as abstract algebras.

Many operations on data structures can be described as abstract algebras.

The most common is a **monoid**, which is any set S with a binary operation $x \cdot y = z$ (x, y, and z are all in S) that

is associative: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

has an identity: there is an $e \in S$ such that $e \cdot x = x$ and $x \cdot e = x$ for all $x \in S$.

Abstract algebra for fun (profit comes later)

Many operations on data structures can be described as abstract algebras.

The most common is a **monoid**, which is any set S with a binary operation $x \cdot y = z$ (x, y, and z are all in S) that

is associative: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

has an identity: there is an $e \in S$ such that $e \cdot x = x$ and $x \cdot e = x$ for all $x \in S$.

(You may be familiar with **groups**, which are **monoids** without inverses.)

Examples of monoids

- Natural numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$) under addition ($+$). The identity is 0.

Examples of monoids

- Natural numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$) under addition ($+$). The identity is 0.
- Positive integers ($\mathbb{P} = \{1, 2, \ldots\}$) under multiplication ($\times$). The identity is 1.

Examples of monoids

- Natural numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$) under addition ($+$). The identity is 0.
- Positive integers ($\mathbb{P} = \{1, 2, \ldots\}$) under multiplication ($\times$). The identity is 1.
- Extended reals ($\mathbb{R} \cup \{-\infty, \infty\}$) under minimization. The identity is ∞.

Examples of monoids

- Natural numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$) under addition ($+$). The identity is 0.
- Positive integers ($\mathbb{P} = \{1, 2, \ldots\}$) under multiplication ($\times$). The identity is 1.
- Extended reals ($\mathbb{R} \cup \{-\infty, \infty\}$) under minimization. The identity is ∞.
- Strings (e.g. `"abc"`, `"def"`) under concatenation (e.g. `"abc" + "def"` $\to$ `"abcdef"`). The identity is the empty string (`""`). (Not commutative!)

Examples of monoids

- Natural numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$) under addition ($+$). The identity is 0.
- Positive integers ($\mathbb{P} = \{1, 2, \ldots\}$) under multiplication ($\times$). The identity is 1.
- Extended reals ($\mathbb{R} \cup \{-\infty, \infty\}$) under minimization. The identity is ∞.
- Strings (e.g. `"abc"`, `"def"`) under concatenation (e.g. `"abc" + "def"` $\to$ `"abcdef"`). The identity is the empty string (`""`). (Not commutative!)
- Histogram contents with identical binning under histogram-addition (`hadd`). The identity is the empty histogram. We can parallelize histogram-filling because adding bin contents is associative: we get the same answer no matter how `fill` operations are divided up among workers.

Examples of monoids

- Natural numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$) under addition ($+$). The identity is 0.
- Positive integers ($\mathbb{P} = \{1, 2, \ldots\}$) under multiplication ($\times$). The identity is 1.
- Extended reals ($\mathbb{R} \cup \{-\infty, \infty\}$) under minimization. The identity is ∞.
- Strings (e.g. `"abc"`, `"def"`) under concatenation (e.g. `"abc" + "def"` $\to$ `"abcdef"`). The identity is the empty string (`""`). (Not commutative!)
- Histogram contents with identical binning under histogram-addition (`hadd`). The identity is the empty histogram. We can parallelize histogram-filling because adding bin contents is associative: we get the same answer no matter how `fill` operations are divided up among workers.
- Boost::Histogram axes under Cartesian product: n axes, an n-dimensional space, combined with m axes, an m-dimensional space, forms an $(n + m)$-dimensional space. An axis with 1 bin could be called an identity.

For superhistograms, we need a semiring

A **semiring** is any set S with two operations, "$+$" and "$\times$", such that

- S under $+$ is a monoid; let's call its identity "0".
- S under $\times$ is a monoid; let's call its identity "1".
- $+$ is commutative: $\boxed{a + b = b + a}$.
- 0 absorbs everything under $\times$: $\boxed{a \times 0 = 0 = 0 \times a}$.
- $\times$ is distributive over $+$: $\boxed{a \times (b + c) = (a \times b) + (a \times c)}$
  and $\boxed{(b + c) \times a = (b \times a) + (c \times a)}$.

For superhistograms, we need a semiring

A **semiring** is any set S with two operations, "$+$" and "$\times$", such that

- S under $+$ is a monoid; let's call its identity "0".
- S under $\times$ is a monoid; let's call its identity "1".
- $+$ is commutative: $\boxed{a + b = b + a}$.
- 0 absorbs everything under $\times$: $\boxed{a \times 0 = 0 = 0 \times a}$.
- $\times$ is distributive over $+$: $\boxed{a \times (b + c) = (a \times b) + (a \times c)}$
  and $\boxed{(b + c) \times a = (b \times a) + (c \times a)}$.

Example: natural numbers under ordinary addition and multiplication.

Superhistograms as a semiring

To build a superhistogram, we put axes together in two ways:

- Cartesian product $\times$, to form a space, like an ordinary Boost::Histogram.
- Collection $+$, to form a set, like a directory of histograms.

Superhistograms as a semiring

To build a superhistogram, we put axes together in two ways:

- Cartesian product $\times$, to form a space, like an ordinary Boost::Histogram.
- Collection $+$, to form a set, like a directory of histograms.

```
StrCategory(["data", "mc"], name="src") * (
    Reg(100, 0, 30, name="pt") + Reg(50, -5, 5, name="eta")
)
```

is equal to

```
StrCategory(["data", "mc"], name="src") * Reg(100, 0, 30, name="pt") +
StrCategory(["data", "mc"], name="src") * Reg(50, -5, 5, name="eta")
```

Two histograms have the same categorical axis, different regular axes.

Checking the properties

- Superhistograms under $+$ is a commutative monoid: a set of histograms has no intrinsic order and all lives at one level (no subdirectories).
- The identity of $+$ is the empty set (no histograms).

Checking the properties

- Superhistograms under $+$ is a commutative monoid: a set of histograms has no intrinsic order and all lives at one level (no subdirectories).
- The identity of $+$ is the empty set (no histograms).
- Superhistograms under $\times$ is a monoid: an n-dimensional space $\times$ an m-dimensional space forms an $(n+m)$-dimensional space.
- The identity of $\times$ is a one-bin axis.

Checking the properties

- Superhistograms under $+$ is a commutative monoid: a set of histograms has no intrinsic order and all lives at one level (no subdirectories).
- The identity of $+$ is the empty set (no histograms).
- Superhistograms under $\times$ is a monoid: an n-dimensional space $\times$ an m-dimensional space forms an $(n+m)$-dimensional space.
- The identity of $\times$ is a one-bin axis.
- $+$ and $\times$ obey a distributive property: if a, b, and c are axes,

$$a \times (b + c) = (a \times b) + (a \times c)$$

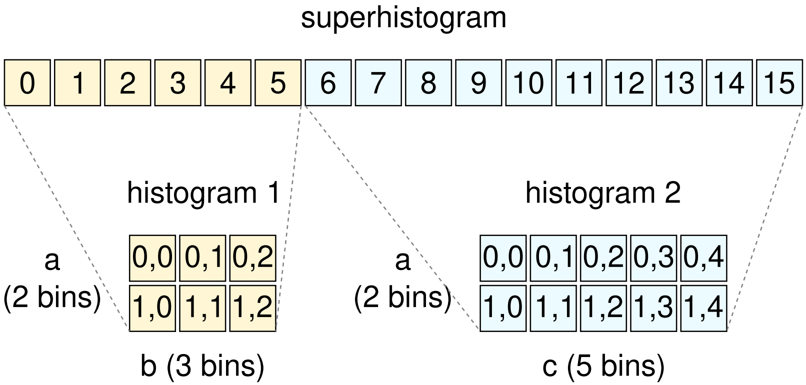
  represents two 2-dimensional histograms, both with the same first axis a, differing in their second axes b or c.

A superhistogram is a jagged array

In Boost::Histogram terms, `Axis` a, b, and c in expressions like

$$a \times (b + c) = (a \times b) + (a \times c)$$

would share one `Storage`, such as `Double` (an array of floating-point values).



A superhistogram is filled by DataFrames

Each `fill` operation increments 1 bin in each multiplicative space, and every such space in an additive collection.

```
h = StrCategory(["data", "mc"], name="src") * (
    Reg(100, 0, 30, name="pt") + Reg(50, -5, 5, name="eta")
)
```

`h.fill(`

	src	pt	eta
0	data	22.1	-1.4
1	data	15.8	2.8
2	data	25.0	0.5
3	mc	19.4	-3.1
4	mc	28.5	-0.7

`)`

fills the histogram with axes `"src"` and `"pt"` 5 times, and
fills the histogram with axes `"src"` and `"eta"` 5 times.

Why does it matter that superhistograms form a semiring?

Superhistograms/semirings have two canonical forms:

- fully expanded: $(a \times b) + (a \times c)$
- maximally factorized: $a \times (b + c)$

Superhistograms/semirings have two canonical forms:

- fully expanded: $(a \times b) + (a \times c)$
- maximally factorized: $a \times (b + c)$

The expanded form is ideal for *filling*. The histograms are in a single, flat collection, a jagged array over the `Storage`.

Finding bin i of histogram n is an $\mathcal{O}(1)$ operation, a single offset-lookup.

Superhistograms/semirings have two canonical forms:

- fully expanded: $(a \times b) + (a \times c)$
- maximally factorized: $a \times (b + c)$

The expanded form is ideal for *filling*. The histograms are in a single, flat collection, a jagged array over the `Storage`.

Finding bin i of histogram n is an $\mathcal{O}(1)$ operation, a single offset-lookup.

The factorized form is ideal for *serialization*. Repeated axis metadata only needs to be stored once. (Important for large variable and categorical axes.)

Superhistograms/semirings have two canonical forms:

- fully expanded: $(a \times b) + (a \times c)$
- maximally factorized: $a \times (b + c)$

The expanded form is ideal for *filling*. The histograms are in a single, flat collection, a jagged array over the `Storage`.

Finding bin i of histogram n is an $\mathcal{O}(1)$ operation, a single offset-lookup.

The factorized form is ideal for *serialization*. Repeated axis metadata only needs to be stored once. (Important for large variable and categorical axes.)

Maybe this could also simplify the interface. (Slicing across all the histograms?)

Contrast to Histogrammar

Histogrammar objects were trees, without a distributive property.

Contrast to Histogrammar

Histogrammar objects were trees, without a distributive property.

They couldn't be expanded or factorized, couldn't share a `Storage`, and generically needed lambda functions to be manually set on each node of the tree to say what the `fill` operation should do.

Contrast to Histogrammar

Histogrammar objects were trees, without a distributive property.

They couldn't be expanded or factorized, couldn't share a `Storage`, and generically needed lambda functions to be manually set on each node of the tree to say what the `fill` operation should do.

By adding this algebraic structure, we restrict the possibilities, but in ways that have benefits to how we want to fill, store, and manipulate histograms.

Onward

To Peter's talk!