

# SuperhistogramS

— or —

## abstract algebra for fun and profit

Jim Pivarski

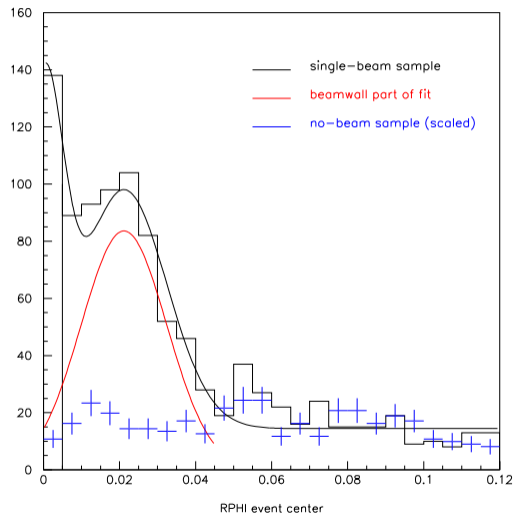
Princeton University – IRIS-HEP

November 15, 2023



# Vague statement of the problem

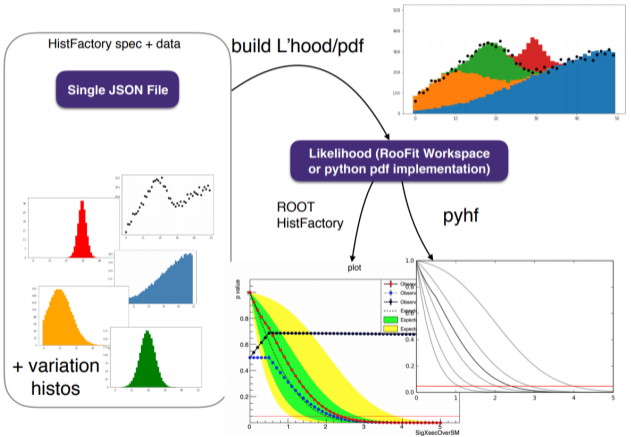
Long, long ago, histograms were individual objects that were managed individually.



# 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.

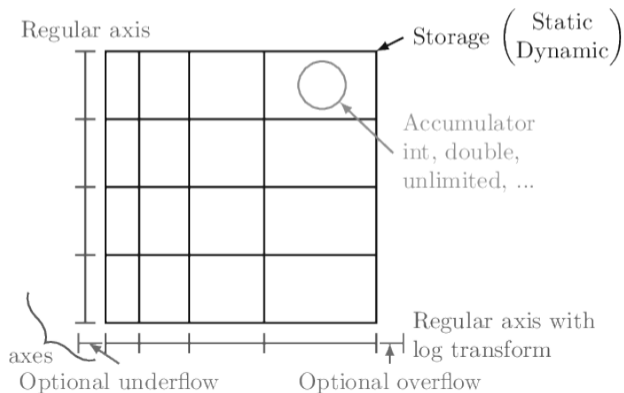


Representing a superhistogram with a directory of ordinary histograms is

- ▶ **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.



```
0          0.5          1
|-----|-----|-----|-----|
|-----|-----|-----|-----|
bh.axis.Regular(10,0,1)
```

```
0      0.3  0.5          1
|-----|-----|-----|-----|
|-----|-----|-----|-----|
bh.axis.Variable([0,.3,.5,1])
```

```
2  5  8  3  7
|-----|-----|-----|-----|
bh.axis.IntCategory([2,5,8,3,7])
```



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.



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.





```
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



## histo·grammar */histō,'g.ræm.ər/*

### GitHub Gists

Informal repository of usable snippets. Gists can be updated more frequently than tutorials.

Search for

histogrammar

### StackOverflow

Question-and-answer site for rapid help. Use the [\[histogrammar\]](#) tag.

Search for

[histogrammar]

### Installation

Getting it.

Download source code from a fixed release

Use Histogrammar's [conda channel](#) (e.g., `conda install -c dianahep histo-grammar`). Below the table, you can find the source code for each release (which includes a table of source code packages) to a zip and a tar.gz.

Fixed releases are stable but not necessarily the latest ones being developed. Each build includes the compiled binaries of that version. Download, if you prefer, the source code from the [source code](#).

Download or clone source code from the bleeding edge

Here is a [link to downloading](#).

To clone the source code:

```
git clone https://github.com/dianahep/histogrammar.git
```

If you use a GitHub user, you can also link the repository to prepare pull requests. If you're looking for a new feature, you may have been directed to [contribute](#).

Install Histogrammar-Python from source

System-wide

Use the standard `Python` install steps.

### Tutorials

How to make plot X from data Y.

Plotting front-ends

Scala

- [Using Scala in Spark](#): These examples describe Spark data in Histogrammar and used to the data plotting package in Scala.

Python

- [Using HISTOGRAMMAR in Spark](#): How to use your Apache Spark cluster to make histograms in Python. This tutorial is available through the [Python](#) package in Python.
- [Using HISTOGRAMMAR in Spark](#): How to use your Apache Spark cluster to make histograms in Python. This tutorial is available through the [Python](#) package in Python.

Aggregation back-ends

Scala

- [Using HISTOGRAMMAR in Spark](#): How to use your Apache Spark cluster to make histograms in Scala. This tutorial is available through the [Scala](#) package in Scala.
- [Using HISTOGRAMMAR in Spark](#): How to use your Apache Spark cluster to make histograms in Scala. This tutorial is available through the [Scala](#) package in Scala.

Python

### Specification

How every primitive should behave in every language.

Bin: regular binning for histograms

Get a user-defined bin width and a user-defined number of bins and a user-defined number of bins and a user-defined number of bins.

What happens with `bin`, the problem is standard Histogrammar

```
bin(100, 0, 10, 100, 100, 100)
```

and when `bin`, it produces a two-dimensional histogram

```
bin(100, 0, 10, 100, 100, 100)
```

Creating with `bin` produces a `python` specific plot?

```
bin(100, 0, 10, 100, 100, 100)
```

and so on.

Binning constructor and required members

- `bin` (constructor) is the number of bins, must be at least one.
- `bin` (constructor) is the minimum value of the bins.



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.



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  $x \in S$ .



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.)



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





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



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



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



- ▶ Natural numbers ( $\mathbb{N} = \{0, 1, 2, \dots\}$ ) under addition ( $+$ ). The identity is 0.
- ▶ Positive integers ( $\mathbb{P} = \{1, 2, \dots\}$ ) under multiplication ( $\times$ ). The identity is 1.
- ▶ Extended reals ( $\mathbb{R} \cup \{-\infty, \infty\}$ ) under minimization. The identity is  $\infty$ .
- ▶ Strings (e.g. "abc", "def") under concatenation (e.g. "abc" + "def"  $\rightarrow$  "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.



- ▶ Natural numbers ( $\mathbb{N} = \{0, 1, 2, \dots\}$ ) under addition ( $+$ ). The identity is 0.
- ▶ Positive integers ( $\mathbb{P} = \{1, 2, \dots\}$ ) under multiplication ( $\times$ ). The identity is 1.
- ▶ Extended reals ( $\mathbb{R} \cup \{-\infty, \infty\}$ ) under minimization. The identity is  $\infty$ .
- ▶ Strings (e.g. "abc", "def") under concatenation (e.g. "abc" + "def"  $\rightarrow$  "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.



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:  $a + b = b + a$ .
- ▶ 0 absorbs everything under  $\times$ :  $a \times 0 = 0 = 0 \times a$ .
- ▶  $\times$  is distributive over  $+$ :  $a \times (b + c) = (a \times b) + (a \times c)$   
and  $(b + c) \times a = (b \times a) + (c \times a)$ .



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:  $a + b = b + a$ .
- ▶ 0 absorbs everything under  $\times$ :  $a \times 0 = 0 = 0 \times a$ .
- ▶  $\times$  is distributive over  $+$ :  
 $a \times (b + c) = (a \times b) + (a \times c)$   
and  $(b + c) \times a = (b \times a) + (c \times a)$ .

**Example:** natural numbers under ordinary addition and multiplication.



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.



- ▶ 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  $+$  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.



- ▶ 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  $c$ .

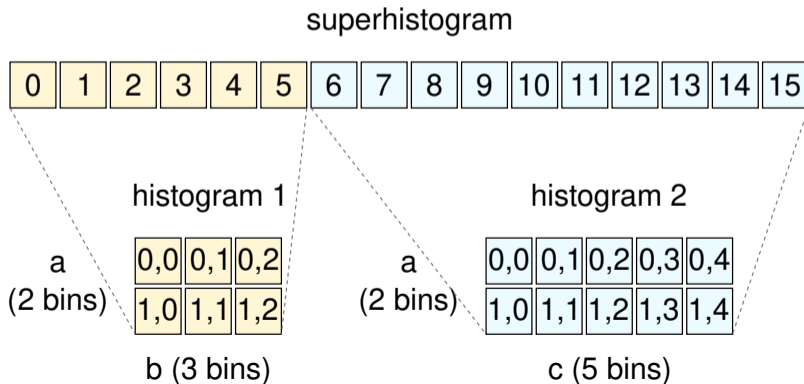


# A superhistogram is a jagged array

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?)



Histogrammar objects were trees, without a distributive property.



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.



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.



To Peter's talk!