histo·*grammar*

/histō,ˈgɹæm.ər/

MAKING HISTOGRAMS FUNCTIONAL

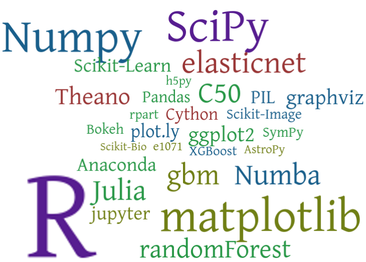
Jim Pivarski

Princeton University – DIANA-HEP

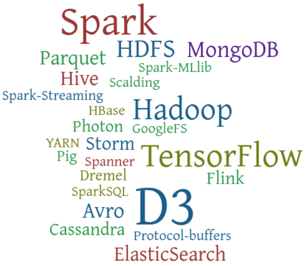
StrangeLoop
September 15, 2016

Tale of two cities

Statistical Computing

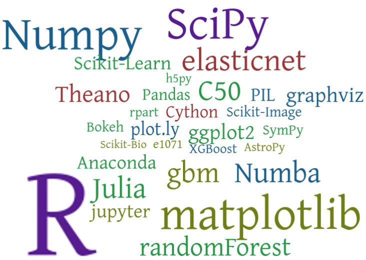


Big Data

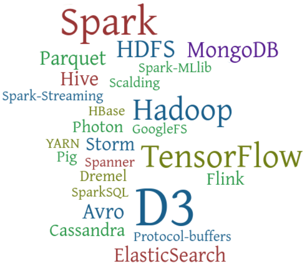


Tale of two cities

Statistical Computing



Big Data



- Mostly natively compiled, driven by high-level languages.
- Primary customer is the laptop data analysis.

Tale of two cities

Statistical Computing

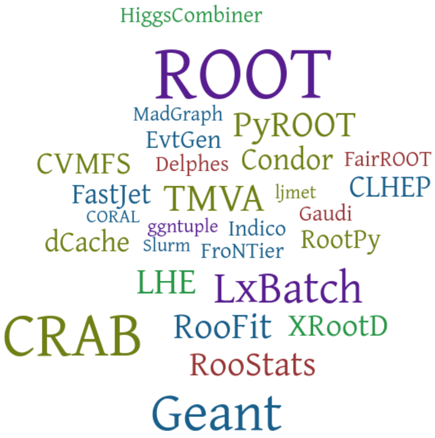
- Mostly natively compiled, driven by high-level languages.
- Primary customer is the laptop data analysis.

Big Data

- Mostly Java/Spark/Clojure.
- More emphasis on scale-out than single-processor speed.
- Datasets assumed to be *big*.

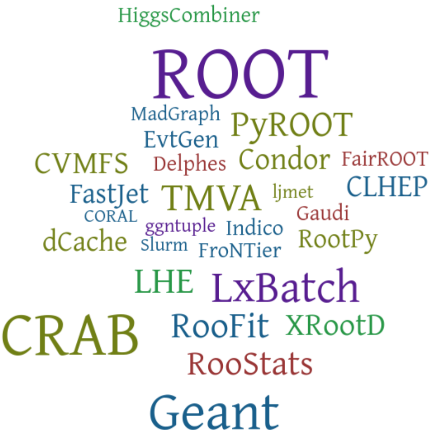
A third you may not have heard about

High Energy Physics (HEP)



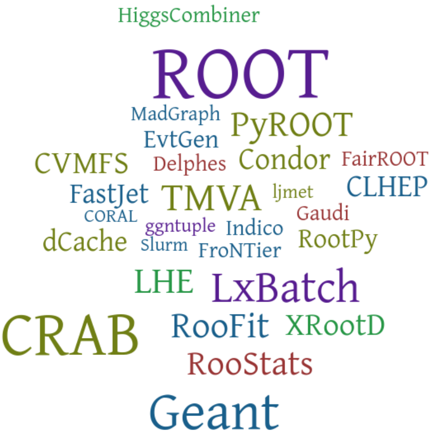
A third you may not have heard about

High Energy Physics (HEP)



- Natively compiled, optimized for single-processor throughput.

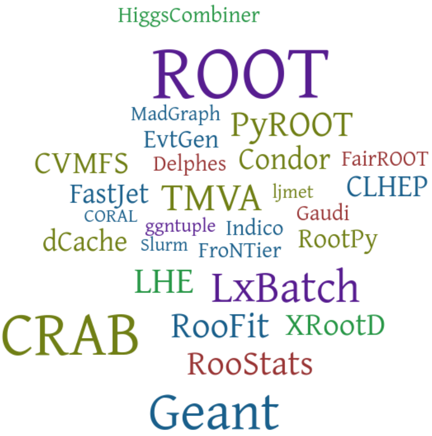
High Energy Physics (HEP)



- Natively compiled, optimized for single-processor throughput.
- *Throughput*, not speed: this is not High Performance Computing (HPC).

A third you may not have heard about

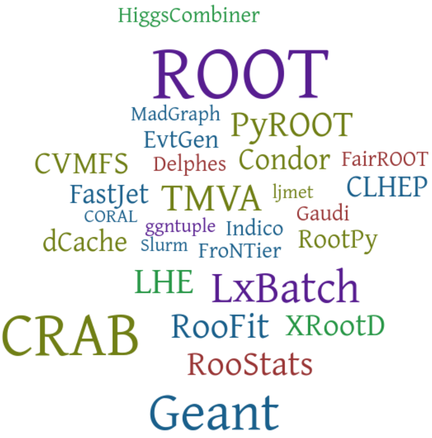
High Energy Physics (HEP)



- Natively compiled, optimized for single-processor throughput.
- *Throughput*, not speed: this is not High Performance Computing (HPC).
- Datasets have always been "big"
  - SPS in 1980's: ∼100 GB per year

A third you may not have heard about

High Energy Physics (HEP)



- Natively compiled, optimized for single-processor throughput.
- *Throughput*, not speed: this is not High Performance Computing (HPC).
- Datasets have always been "big"
  - SPS in 1980's: $\sim$100 GB per year
  - LHC today: $\sim$25 PB per year

For decades, HEP seemed to be the only field with these problems.

For decades, HEP seemed to be the only field with these problems.

That is no longer true.

For decades, HEP seemed to be the only field with these problems.

That is no longer true.

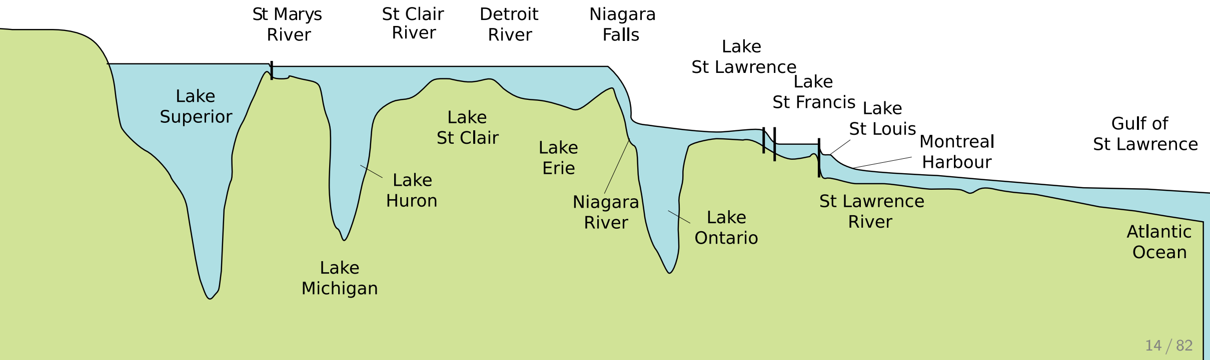
- HEP software needs clearly overlap with the Scipy/R/ Scikit-Learn world and the Spark/Hadoop/NoSQL world.

For decades, HEP seemed to be the only field with these problems.

That is no longer true.

- HEP software needs clearly overlap with the Scipy/R/Scikit-Learn world and the Spark/Hadoop/NoSQL world.
- Individual physicists and projects like DIANA-HEP (my employer) are starting to explore and develop these connections.

Considering how much has been developed on both sides of the divide, small "glue projects" connecting them can have a big impact.



Taxonomy of the Revolution

Type I: HEP software that serves *the same function* as software in the wider community.

Type II: Domain-specific software for HEP applications. For example, "HiggsCombiner."

Type III: HEP software and concepts that would benefit the wider community.

Taxonomy of the Revolution

Type I: HEP software that serves *the same function* as software in the wider community.

Wider community has better resources for

- maintaining code
- catching bugs
- revising bad designs.

Type II: Domain-specific software for HEP applications. For example, "HiggsCombiner."

Type III: HEP software and concepts that would benefit the wider community.

Taxonomy of the Revolution

Type I: HEP software that serves *the same function* as software in the wider community.

Wider community has better resources for

- maintaining code
- catching bugs
- revising bad designs.

Type II: Domain-specific software for HEP applications. For example, "HiggsCombiner."

Obviously. This really is a unique problem.

Type III: HEP software and concepts that would benefit the wider community.

Type I: HEP software that serves *the same function* as software in the wider community.

Wider community has better resources for

- maintaining code
- catching bugs
- revising bad designs.

Type II: Domain-specific software for HEP applications. For example, "HiggsCombiner."

Obviously. This really is a unique problem.

Type III: HEP software and concepts that would benefit the wider community.

Cultural exchange goes in both directions.

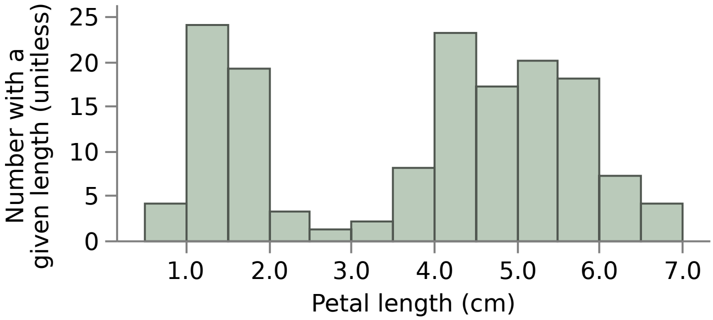
Histograms are an example of Type III:

HEP has a unique and useful approach.

Statistical definition of a histogram

A **histogram** is an approximation of a distribution, formed by partitioning a sample by one of its features and counting how many items fall in each partition.

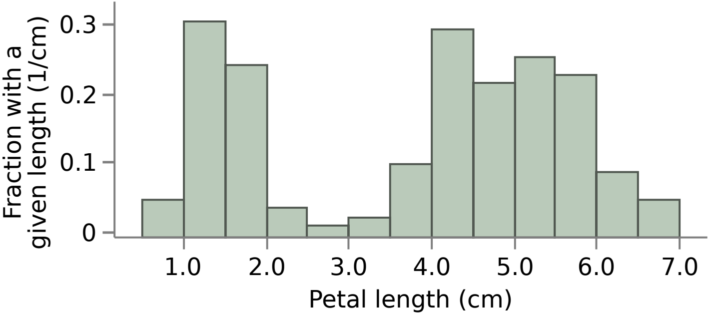
The partitions are called **bins**, and the content of each bin may be represented as the total count or the average density in that partition. [Start philosophical argument here.]



Statistical definition of a histogram

A **histogram** is an approximation of a distribution, formed by partitioning a sample by one of its features and counting how many items fall in each partition.

The partitions are called **bins**, and the content of each bin may be represented as the total count or the average density in that partition. [Start philosophical argument here.]



Gallery of histogram APIs: non-HEP

R: `hist(x, breaks = "Sturges", ...)`

`x` input data samples
`breaks` binning strategy

Numpy: `numpy.histogram(a, bins = 10, ...)`

`a` input data samples
`bins` binning strategy

Pandas: `DataFrame.hist(data, bins = 10, ...)`

`data` input data samples
`bins` binning strategy

Spark: `DoubleRDDFunctions.histogram(buckets: Array[Double])`

`this` input data samples
`buckets` binning strategy

Mathematica: `Histogram[data, hspec]`

`data` input data samples
`hspec` binning strategy

MATLAB: `histogram(X, nbins)`

`X` input data samples
`nbins` binning strategy

Gallery of histogram APIs: non-HEP

R: `hist(x, breaks = "Sturges", ...)`

`x` input data samples
`breaks` binning strategy

Nu

`bins` binning strategy

Pandas: `DataFrame.hist(data, bins = 10, ...)`

`data` input data samples
`bins` binning strategy

Spark: `DoubleRDDFunctions.histogram(buckets: Array[Double])`

les
tegy

`hspec]`

les
`hspec` binning strategy

MATLAB: `histogram(X, nbins)`

`X` input data samples
`nbins` binning strategy

They all require *all* of the input data before giving you a histogram.

HEP histograms are different

HEP software (HBOOK, PAW, ROOT, HippoDraw, AIDA, . . .) treats histograms as *fillable containers.*

```
h = Histogram(numBins,
        lowEdge, highEdge)

for x in data:
    h.fill(x)

h.plot()
```

API presumes the dataset is too big for a single function call.

Intrinsically single-pass (implementation can't pre-scan to set bin width or range).

HEP histograms are different

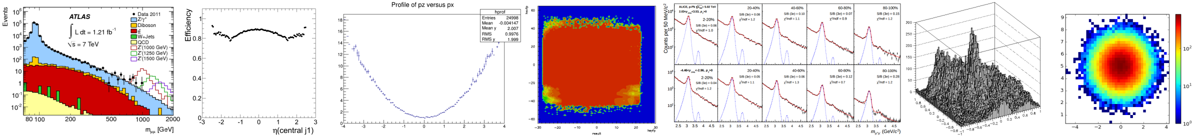
HEP software (HBOOK, PAW, ROOT, HippoDraw, AIDA, . . .) treats histograms as *fillable containers.*

```
h = Histogram(numBins,
        lowEdge, highEdge)

for x in data:
    h.fill(x)

h.plot()
```

(HBOOK has been doing this for 43 years.)

API presumes the dataset is too big for a single function call.

Intrinsically single-pass (implementation can't pre-scan to set bin width or range).

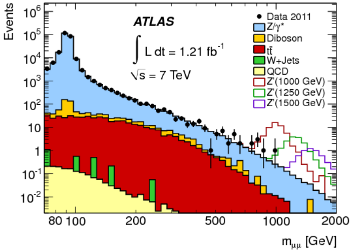


Histogram containers are the basic unit of HEP data analysis, used to make just about everything else. Analogous to

- lists in LISP
- dictionaries in Python
- data.frames in R

Examples (only one of which is really a histogram)

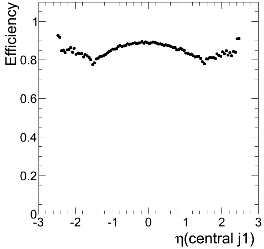
Stacked histograms



Represents contributions from different samples to a total histogram.

Constructed by cumulatively filling a series of histograms and overlaying them in reverse order.

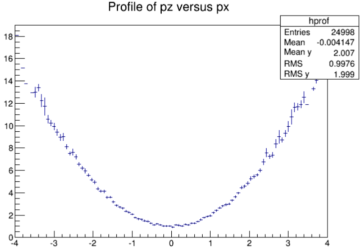
Efficiency plot



Represents the probability of passing a filter versus some variable.

Constructed by filling two histograms, one with the filter, the other without, and dividing them bin-by-bin.

Profile plot



Represents a marginal projection of the dataset with errors on the mean.

Constructed by filling $\sum_i y_i$ and $\sum_i {y_i}^2$ separately and doing the appropriate transformations bin-by-bin.

The problem with this API

Domain-specific knowledge enters in two different places: histogram-construction and histogram-filling.

construction
```
job[N].h = Histogram(numBins, lowEdge, highEdge)
               # binning requires knowledge of the problem domain
```

filling
```
for event in job[N].physicsEvents:
    job[N].h.fill(event.whatToPlot())
               # whatToPlot() requires knowledge of the problem domain
```

merging
```
h = job[0].h + job[1].h + job[2].h + ...
```

Specifically, in Apache Spark. . .

Imperative analysis script

```
x = Histogram(100, -5.0, 5.0)

for event in events:
    x.fill(event.calcX())

x.plot()
```

Spark equivalent

```
x = events.aggregate(
    emptyCounter(),
    lambda h, event:
        incrementCounter(h, event),
    lambda h1, h2:
        addCounters(h1, h2))

x.plot()
```

Specifically, in Apache Spark. . .

Imperative analysis script

```
x = Histogram(100, -5.0, 5.0)

for event in events:
    x.fill(event.calcX())

x.plot()
```

Spark equivalent

```
x = events.aggregate(
    Histogram(100, -5.0, 5.0),
    lambda h, event:
        h.fill(event.calcX()),
    lambda h1, h2:
        h1 + h2)

x.plot()
```

Specifically, in Apache Spark. . .

Imperative analysis script

```
x = Histogram(100, -5.0, 5.0)
y = Histogram(100, -5.0, 5.0)

for event in events:
    x.fill(event.calcX())
    y.fill(event.calcY())

x.plot()
y.plot()
```

Spark equivalent

```
x, y = events.aggregate(
    (Histogram(100, -5.0, 5.0),
     Histogram(100, -5.0, 5.0)),
    lambda hs, event: (
        hs[0].fill(event.calcX()),
        hs[1].fill(event.calcY())),
    lambda hs1, hs2: (
        hs1[0] + hs2[0],
        hs1[1] + hs2[1]))

x.plot()
y.plot()
```

Specifically, in Apache Spark. . .

Imperative analysis script

```
x = Histogram(100, -5.0, 5.0)
y = Histogram(100, -5.0, 5.0)
z = Histogram(100, -5.0, 5.0)

for event in events:
    x.fill(event.calcX())
    y.fill(event.calcY())
    z.fill(event.calcZ())

x.plot()
y.plot()
z.plot()
```

Spark equivalent

```
x, y, z = events.aggregate(
    (Histogram(100, -5.0, 5.0),
     Histogram(100, -5.0, 5.0),
     Histogram(100, -5.0, 5.0)),
    lambda hs, event: (
        hs[0].fill(event.calcX()),
        hs[1].fill(event.calcY()),
        hs[2].fill(event.calcZ())),
    lambda hs1, hs2: (
        hs1[0] + hs2[0],
        hs1[1] + hs2[1],
        hs1[2] + hs2[2]))

x.plot()
y.plot()
z.plot()
```

Solution: go functional

Make the constructor a higher-order function:

```
h = Histogram(numBins, lowEdge, highEdge, fillRule)
```

where **`fillRule`** is a function : *data* $\to \mathbb{R}$ that determines which bin an element of *data* increments.

Solution: go functional

Make the constructor a higher-order function:

```
h = Histogram(numBins, lowEdge, highEdge, fillRule)
```

where **`fillRule`** is a function : *data* $\to \mathbb{R}$ that determines which bin an element of *data* increments.

All domain-specific knowledge is in the constructor. The filling function may now be generic (and automated).

```
h.fill(datum)        # calls fillRule(datum) internally
```

Solution: go functional

Make the constructor a higher-order function:

```
h = Histogram(numBins, lowEdge, highEdge, fillRule)
```

where **`fillRule`** is a function : *data* $\to \mathbb{R}$ that determines which bin an element of *data* increments.

All domain-specific knowledge is in the constructor. The filling function may now be generic (and automated).

```
h.fill(datum)        # calls fillRule(datum) internally
```

(In a *purely* functional environment, `newh = oldh.filled(datum)`.)

What this looks like in Spark

```python
# Define increment and combine
# as reusable library functions.

def increment(h, event):
    h.fill(event)
    return h

def combine(h1, h2):
    return h1 + h2

x = events.aggregate(
    Histogram(100, -5.0, 5.0,
        lambda ev: ev.calcX()),
    increment,
    combine)


x.plot()
```

What this looks like in Spark

```
# Define increment and combine
# as reusable library functions.

def increment(h, event):
    h.fill(event)
    return h

def combine(h1, h2):
    return h1 + h2

x = events.aggregate(
    Histogram(100, -5.0, 5.0,
       lambda ev: ev.calcX()),
    increment,
    combine)


x.plot()
```

```
class Label:      # also in the library
  def __init__(self, **hs):
      self.hs = hs
  def fill(self, datum):
      for h in self.hs.values():
          h.fill(datum)
  def __add__(self, other):
      return {n: self.hs[n] + other.hs[n]
                        for n in self.hs}

pkg = events.aggregate(Label(
        x = Histogram(100, -5.0, 5.0,
                lambda ev: ev.calcX()),
        y = Histogram(100, -5.0, 5.0,
                lambda ev: ev.calcY()))
    increment, combine)

pkg.hs["x"].plot()
```

This packaging with "`Label`" has the same interface as a `Histogram`, namely the `fill` and `+` methods.

Histograms and collections of histograms are now interchangeable.

Generic aggregators:

Count	number of times "fill" is called
Bin$[\cdot]$	partitions according to fillRule, passes data to *one* subaggregator
Label$[\cdot]$	passes data to *all* named subaggregators

(Naming convention: all aggregators are verbs.)

With the right set of aggregators, you can do anything

Histograms:

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

Two-dimensional histograms:

```
Bin(xnum, xlow, xhigh, xfill,
  Bin(ynum, ylow, yhigh, yfill,
    Count()))
```

Profile plots:

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

where `Deviate` aggregates a mean and standard deviation.

Mix and match binning methods:

```
IrregularlyBin([-2.4, -2.1, -1.5,
    0.0, 1.5, 2.1, 2.4],
  filleta,
  Bin(314, -3.14, 3.14, fillphi,
    Count()))

SparselyBin(0.01, filleta,
  Bin(314, -3.14, 3.14, fillphi,
    Count()))

Categorize(fillByName,
  Bin(314, -3.14, 3.14, fillphi,
    Count()))
```

With the right set of aggregators, you can do anything

Create complex trees, detailing exactly what you want to aggregate:

```
Bin(xnum, xlow, xhigh, lambda datum: datum.x,
  Branch(Count(),
         Minimize(lambda datum: datum.y),
         Maximize(lambda datum: datum.y),
         Average(lambda datum: datum.y),
         Sum(lambda datum: datum.weight),
         Sum(lambda datum: datum.weight**2)))
```

Now each bin in `x` contains

- number of entries,
- minimum `y` value,
- maximum `y` value,
- average of `y`,
- sum of weights,
- sum of weights squared.

With the right set of aggregators, you can do anything

High-level interface to common patterns:

```
Fraction(cut, Bin(numBins, lowEdge, highEdge, fillRule))
```

to make a ratio plot,

```
Stack(cuts, Bin(numBins, lowEdge, highEdge, fillRule))
```

to make stacked histograms.

Guarantees that the bins align in the numerator/denominator or stacked items, so they can be correctly divided or added together.

dianahep DOCUMENTATION GITHUB

histo·*grammar*

/histō,ˈgɹæm.ər/

MAKING HISTOGRAMS FUNCTIONAL

Histogrammar is a suite of data aggregation primitives for making histograms and much, much more. A few composable functions can generate many different types of plots, and these functions are reimplemented (exactly!) in multiple languages and serialized to JSON for cross-platform compatibility.

Histogrammar

Histogrammar is a suite of composable aggregators with

Histogrammar

Histogrammar is a suite of composable aggregators with

- a language-independent specification,

Histogrammar

Histogrammar is a suite of composable aggregators with

- a language-independent specification,
- several language versions (Python and Scala are the most complete),

Histogrammar is a suite of composable aggregators with

- a language-independent specification,
- several language versions (Python and Scala are the most complete),
- an interchangeable JSON format,

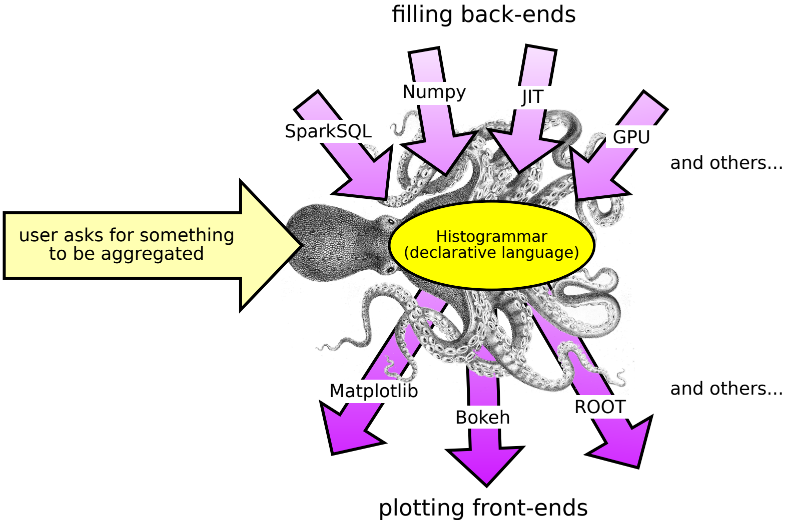
Histogrammar is a suite of composable aggregators with

- a language-independent specification,
- several language versions (Python and Scala are the most complete),
- an interchangeable JSON format,
- multiple filling back-ends (examples follow),

Histogrammar is a suite of composable aggregators with

- a language-independent specification,
- several language versions (Python and Scala are the most complete),
- an interchangeable JSON format,
- multiple filling back-ends (examples follow),
- no built-in plotting: Matplotlib, Bokeh, ROOT as front-ends.

User's perspective



Current coverage

Aggregator	Scala	JVM-JIT	Python	Numpy	C-JIT	GPU	C++11	Julia	R	Javascript
Count	DONE		DONE	DONE	DONE	DONE	trial	DONE		
Sum	DONE		DONE	DONE	DONE	DONE	trial	DONE		
Average	DONE		DONE	DONE	DONE	DONE		DONE		
Deviate	DONE		DONE	DONE	DONE	DONE		DONE		
Minimize	DONE		DONE	DONE	DONE	DONE		DONE		
Maximize	DONE		DONE	DONE	DONE	DONE		DONE		
Bag	DONE		DONE	DONE	DONE	growable?		DONE		
Bin	DONE		DONE	DONE	DONE	DONE	trial	DONE		
SparselyBin	DONE		DONE	DONE	DONE	growable?				
CentrallyBin	DONE		DONE	DONE	DONE	DONE				
IrregularlyBin	DONE		DONE	DONE	DONE	DONE				
Categorize	DONE		DONE	DONE	DONE	growable?				
Fraction	DONE		DONE	DONE	DONE	DONE				
Stack	DONE		DONE	DONE	DONE	DONE				
Select	DONE		DONE	DONE	DONE	DONE	trial			
Label	DONE		DONE	DONE	DONE	DONE				
UntypedLabel	DONE		DONE	DONE	DONE	DONE				
Index	DONE		DONE	DONE	DONE	DONE				
Branch	DONE		DONE	DONE	DONE	DONE				

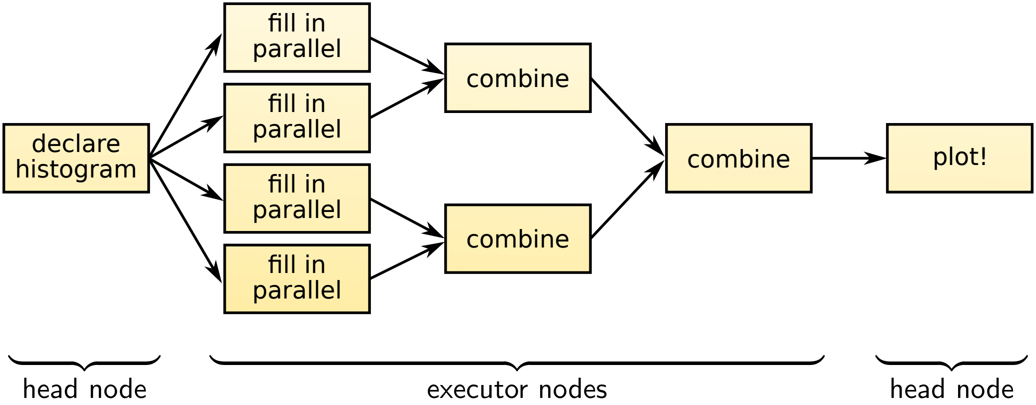
The "Bag" specification (multiset for scatter-plots) may be revised soon.

The C++ implementation *will* be revised.

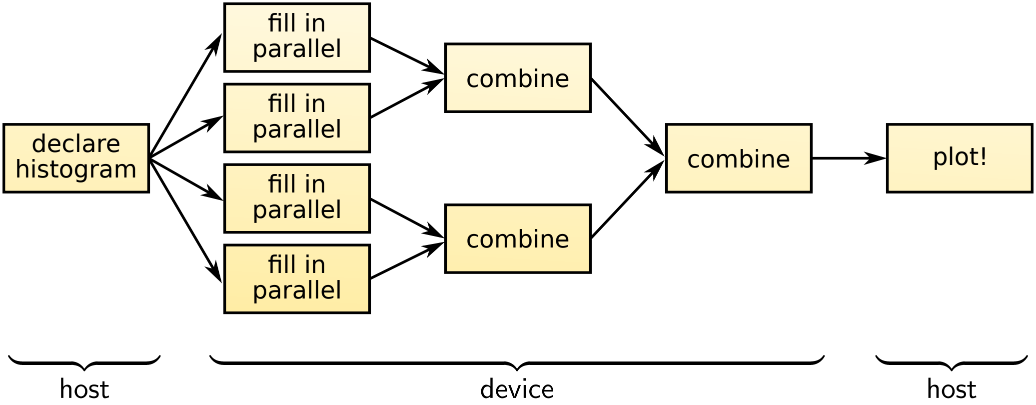
Designed for distributed workflows

dianahep

Parallel histogramming in Spark



Parallel histogramming on a GPU



diana hep — Installation | Tutorials | Specification | References | GitHub repos

Bin: regular binning for histograms

Split a quantity into equally spaced bins between a low and high threshold and fill exactly one bin per datum.

When composed with Count, this produces a standard histogram:

```
Bin.ing(100, 0, 10, fill_x, Count.ing())
```

and when nested, it produces a two-dimensional histogram:

```
Bin.ing(100, 0, 10, fill_x,
  Bin.ing(100, 0, 10, fill_y, Count.ing()))
```

Combining with Deviate produces a physicist's "profile plot:"

```
Bin.ing(100, 0, 10, fill_x, Deviate.ing(fill_y))
```

and so on.

Screenshots from the specification
http://histogrammar.org/docs/specification

Binning constructor and required members

```
Bin.ing(num, low, high, quantity, value=Count.ing(), underflow=Count.ing(), overflow=Count.ing(),
nanflow=Count.ing())
```

- `num` (32-bit integer) is the number of bins; must be at least one.

Binning constructor and required members

```
Bin.ing(num, low, high, quantity, value=Count.ing(), underflow=Count.ing(), overflow=Count.ing(),
nanflow=Count.ing())
```

- `num` (32-bit integer) is the number of bins; must be at least one.
- `low` (double) is the minimum-value edge of the first bin.
- `high` (double) is the maximum-value edge of the last bin; must be strictly greater than `low`.
- `quantity` (function returning double) computes the quantity of interest from the data.
- `value` (present-tense aggregator) generates sub-aggregators to put in each bin.
- `underflow` (present-tense aggregator) is a sub-aggregator to use for data whose quantity is less than `low`.
- `overflow` (present-tense aggregator) is a sub-aggregator to use for data whose quantity is greater than or equal to `high`.
- `nanflow` (present-tense aggregator) is a sub-aggregator to use for data whose quantity is NaN.
- `entries` (mutable double) is the number of entries, initially 0.0.
- `values` (list of present-tense aggregators) are the sub-aggregators in each bin.

Binned constructor and required members

```
Bin.ed(low, high, entries, values, underflow, overflow, nanflow)
```

Fill and combine algorithms

```
def fill(binning, datum, weight):
    if weight > 0.0:
        q = binning.quantity(datum)
        if math.isnan(q):
            fill(binning.nanflow, datum, weight)
        elif q < binning.low:
            fill(binning.underflow, datum, weight)
        elif q >= binning.high:
            fill(binning.overflow, datum, weight)
        else:
            bin = int(math.floor(binning.num * \
                (q - binning.low) / (binning.high - binning.low)))
            fill(binning.values[bin], datum, weight)
        binning.entries += weight

def combine(one, two):
    if one.num != two.num or one.low != two.low or one.high != two.high:
        raise Exception
    entries = one.entries + two.entries
    values = [combine(x, y) for x, y in zip(one.values, two.values)]
    underflow = combine(one.underflow, two.underflow)
    overflow = combine(one.overflow, two.overflow)
    nanflow = combine(one.nanflow, two.nanflow)
    return Bin.ed(one.low, one.high, entries, values, underflow, overflow, nanflow)
```

JSON fragment format

JSON object containing

- `low` (JSON number)
- `high` (JSON number)
- `entries` (JSON number or “inf”)
- `values:type` (JSON string), name of the values sub-aggregator type
- `values` (JSON array of sub-aggregators)
- `underflow:type` (JSON string), name of the underflow sub-aggregator type
- `underflow` sub-aggregator
- `overflow:type` (JSON string), name of the overflow sub-aggregator type
- `overflow` sub-aggregator
- `nanflow:type` (JSON string), name of the nanflow sub-aggregator type
- `nanflow` (sub-aggregator)
- optional `name` (JSON string), name of the `quantity` function, if provided.
- optional `values:name` (JSON string), name of the `quantity` function used by each value. If specified here, it is *not* specified in all the values, thereby streamlining the JSON.

Examples:

Here is a five-bin histogram, whose bin centers are at -4, -2, 0, 2, and 4. It counts the number of measurements made at each position.

```
{"version": "0.9",
 "type": "Bin",
 "data": {
   "low": -5.0,
   "high": 5.0,
   "entries": 123.0,
   "name": "position [cm]",
   "values:type": "Count",
   "values": [10.0, 20.0, 20.0, 30.0, 30.0],
   "underflow:type": "Count",
   "underflow": 5.0,
   "overflow:type": "Count",
   "overflow": 8.0,
   "nanflow:type": "Count",
   "nanflow": 0.0}}
```

Here is another five-bin histogram on the same domain, this one quantifying an average value in each bin. The quantity measured by the average has a name (`"average time [s]"`), which would have been a `"name"` field in the JSON objects representing the averages if it had not been specified once in `"values:name"`.

```
{"version": "0.9",
 "type": "Bin",
 "data": {
```

dianahep | Installation | Tutorials | Specification | References | GitHub repos

Plotting front-ends

Scala

- Making Bokeh plots in Spark: How to aggregate
  plotting package in Scala.

Python

- Making PyROOT plots: How to send Histogramm
  is complete enough that you could start here.
- Making Bokeh plots: How to send Histogrammar data to the Bokeh plotting package in Python. 0%

Tutorials page

http://histogrammar.org/docs/tutorials

Aggregation back-ends

Scala

- Collecting data in Spark: How to use your Apache Spark cluster to make histograms, rather than downloading the data and plotting locally. 100%
- Enhancements for SparkSQL: Special bindings to make histograms directly from Apache SparkSQL tables. 100%

Python

- Enhancements for Numpy: Aggregating over data in Numpy arrays without a Python for loop (i.e. faster). 100%

Spark RDD and DataFrame back-ends

Spark RDD: provide an opaque function

```
import org.dianahep.histogrammar._
val h = rdd.aggregate(
  Bin(10, 0, 100, {mu: Muon => Math.sqrt(mu.px**2 + mu.py**2)}))
   (new Increment, new Combine)
```

(Statically typed to function arguments, hidden by type inference.)

SparkSQL: provide a transformation on columns

```
import org.dianahep.histogrammar.sparksql._
val h = df.histogrammar(Bin(10, 0, 100, sqrt($"px"**2 + $"py"**2)))
```

Python and Numpy back-end

Python: provide a lambda or a quoted expression

```
from math import sqrt
from histogrammar import *
h = Bin(10, 0, 100, lambda muon: sqrt(muon.px**2 + muon.py**2))
h = Bin(10, 0, 100, "sqrt(px**2 + py**2)")
for mu in muons:
    h.fill(mu)
```

Numpy: provide a lambda or a quoted expression

```
from numpy import sqrt
h = Bin(10, 0, 100, lambda muons:
                        sqrt(muons["px"]**2 + muons["py"]**2))
h = Bin(10, 0, 100, "np.sqrt(px**2 + py**2)")
h.fill.numpy(muons)
```

Python and Numpy back-end

Python: provide a lambda or a quoted expression

```
from math import sqrt
from histogrammar import *
h = Bin(10, 0, 100, lambda muon: sqrt(muon.px**2 + muon.py**2))
h = Bin(10, 0, 100, "sqrt(px**2 + py**2)")
for mu in muons:
    h.fill(mu)
```

Numpy: provide a lambda or a quoted expression

```
from numpy import sqrt
h = Bin(10, 0, 100, lambda muons:
                        sqrt(muons["px"]**2 + muons["py"]**2))
h = Bin(10, 0, 100, "np.sqrt(px**2 + py**2)")
h.fill.numpy(muons)
```

Speedups range from nothing (string handling) to 100 times (depending on specific case).

JIT compilation

```
h = Bin(10, 0, 100, "sqrt(px*px + py*py)")    # C code
h.fill.root(muons)
```

1. Walk tree of aggregators to generate (cache-aware) C code instead of filling.
2. Compile the C code (in ROOT with LLVM).
3. Evaluate on a large dataset (in ROOT).
4. Update the original aggregator.

JIT compilation

```
h = Bin(10, 0, 100, "sqrt(px*px + py*py)")    # C code
h.fill.root(muons)
```

1. Walk tree of aggregators to generate (cache-aware) C code instead of filling.
2. Compile the C code (in ROOT with LLVM).
3. Evaluate on a large dataset (in ROOT).
4. Update the original aggregator.

Consistently 100 times faster than Python (and 30% faster than generic C++ code).

GPU code generation

Special case of JIT: generates parallel CUDA code.

```
h = Bin(10, 0, 100, "sqrt(px*px + py*py)")
                                    # expression in C

print h.cuda()                      # prints CUDA code

h.fill.pycuda(muons)                # evaluates with PyCUDA
```

GPU code generation

Special case of JIT: generates parallel CUDA code.

```
h = Bin(10, 0, 100, "sqrt(px*px + py*py)")
                                # expression in C

print h.cuda()                  # prints CUDA code

h.fill.pycuda(muons)            # evaluates with PyCUDA
```

- Not intended as an accelerator (due to cost of sending data to GPU).

GPU code generation

Special case of JIT: generates parallel CUDA code.

```
h = Bin(10, 0, 100, "sqrt(px*px + py*py)")
                               # expression in C

print h.cuda()                 # prints CUDA code

h.fill.pycuda(muons)           # evaluates with PyCUDA
```

- Not intended as an accelerator (due to cost of sending data to GPU).
- Intended as a convenience for GPU algorithm developers.

GPU code generation

Special case of JIT: generates parallel CUDA code.

```
h = Bin(10, 0, 100, "sqrt(px*px + py*py)")
                        # expression in C

print h.cuda()          # prints CUDA code

h.fill.pycuda(muons)    # evaluates with PyCUDA
```

- Not intended as an accelerator (due to cost of sending data to GPU).
- Intended as a convenience for GPU algorithm developers.
- Histogrammar generates a
  - `__device__ fill` function that can be called by GPU code,
  - `__global__ combine` function that merges partial results,
  - `__host__ toJson` function for plotting (e.g. in Python),

to help GPU developers visualize intermediate quantities for debugging.

Lessons learned. . .

The unreasonable effectiveness of mathematics

I learned (along the way) that all aggregators must

The unreasonable effectiveness of mathematics

I learned (along the way) that all aggregators must

be additive:

independent of *whether* datasets are partitioned.

$$\mathrm{fill}(\mathrm{data}_1 + \mathrm{data}_2) = \mathrm{fill}(\mathrm{data}_1) + \mathrm{fill}(\mathrm{data}_2)$$

The unreasonable effectiveness of mathematics

I learned (along the way) that all aggregators must

be additive:
independent of *whether* datasets are partitioned.

$$\mathrm{fill}(\mathrm{data}_1 + \mathrm{data}_2) = \mathrm{fill}(\mathrm{data}_1) + \mathrm{fill}(\mathrm{data}_2)$$

be homogeneous in the weights:
fill weight 0.0 corresponds to no fill, 1.0 to simple fill, 2.0 to double-fill, ...

$$\mathrm{fill}(\mathrm{data}, \mathrm{weight}) = \mathrm{fill}(\mathrm{data}) \cdot \mathrm{weight}$$

I learned (along the way) that all aggregators must

be additive:

independent of *whether* datasets are partitioned.

$$\mathrm{fill}(\mathrm{data}_1 + \mathrm{data}_2) = \mathrm{fill}(\mathrm{data}_1) + \mathrm{fill}(\mathrm{data}_2)$$

be homogeneous in the weights:

fill weight 0.0 corresponds to no fill, 1.0 to simple fill, 2.0 to double-fill, . . .

$$\mathrm{fill}(\mathrm{data}, \mathrm{weight}) = \mathrm{fill}(\mathrm{data}) \cdot \mathrm{weight}$$

linear

The unreasonable effectiveness of mathematics

I learned (along the way) that all aggregators must

be additive:

independent of *whether* datasets are partitioned.

$$\mathrm{fill}(\mathrm{data}_1 + \mathrm{data}_2) = \mathrm{fill}(\mathrm{data}_1) + \mathrm{fill}(\mathrm{data}_2)$$

be homogeneous in the weights:

fill weight 0.0 corresponds to no fill, 1.0 to simple fill, 2.0 to double-fill, . . .

$$\mathrm{fill}(\mathrm{data}, \mathrm{weight}) = \mathrm{fill}(\mathrm{data}) \cdot \mathrm{weight}$$

linear

be associative:

independent of *where* datasets get partitioned.

$$(h_1 + h_2) + h_3 = h_1 + (h_2 + h_3)$$

The unreasonable effectiveness of mathematics

I learned (along the way) that all aggregators must

be additive:

independent of *whether* datasets are partitioned.

$$\mathrm{fill}(\mathrm{data}_1 + \mathrm{data}_2) = \mathrm{fill}(\mathrm{data}_1) + \mathrm{fill}(\mathrm{data}_2)$$

be homogeneous in the weights:

fill weight 0.0 corresponds to no fill, 1.0 to simple fill, 2.0 to double-fill, ...

$$\mathrm{fill}(\mathrm{data}, \mathrm{weight}) = \mathrm{fill}(\mathrm{data}) \cdot \mathrm{weight}$$

linear

be associative:

independent of *where* datasets get partitioned.

$$(h_1 + h_2) + h_3 = h_1 + (h_2 + h_3)$$

have an identity:

for both the fill and $+$ methods.

$$h + 0 = h, \quad 0 + h = h, \quad \mathrm{fill}(\mathrm{data}, 0) = 0$$

I learned (along the way) that all aggregators must

be additive:
independent of *whether* datasets are partitioned.

$$\mathrm{fill}(\mathrm{data}_1 + \mathrm{data}_2) = \mathrm{fill}(\mathrm{data}_1) + \mathrm{fill}(\mathrm{data}_2)$$

be homogeneous in the weights:
fill weight 0.0 corresponds to no fill, 1.0 to simple fill, 2.0 to double-fill, ...

$$\mathrm{fill}(\mathrm{data}, \mathrm{weight}) = \mathrm{fill}(\mathrm{data}) \cdot \mathrm{weight}$$

linear

be associative:
independent of *where* datasets get partitioned.

$$(h_1 + h_2) + h_3 = h_1 + (h_2 + h_3)$$

have an identity:
for both the fill and $+$ methods.

$$h + 0 = h, \quad 0 + h = h, \quad \mathrm{fill}(\mathrm{data}, 0) = 0$$

monoid

RIP: AdaptivelyBin

Want: aggregator that determines binning from data.

Tried: each new datapoint is a new bin; merge closest bins.

Ben-Haim and Tom-Tov, "A streaming parallel decision tree algorithm" *J. Machine Learning Research 11 (2010).*

Problem: result of `fill` depends on history of data seen; not a linear monoid.

Any ideas?

RIP: AdaptivelyBin

Want: aggregator that determines binning from data.

Tried: each new datapoint is a new bin; merge closest bins.

Ben-Haim and Tom-Tov, "A streaming parallel decision tree algorithm" *J. Machine Learning Research 11 (2010).*

Problem: result of `fill` depends on history of data seen; not a linear monoid.

Any ideas?

RIP: Quantile

Want: approximate median or other quantiles.

Tried: incrementally minimize mean absolute error from target quantile.

Problem: result of `fill` depends on history of data seen; not a linear monoid.

Any ideas?

Want to get involved?

http://histogrammar.org

- Tutorials
- Specification
- GitHub organization
- Maven Central/PyPI installation
- Scaladocs/Sphinx reference

Want to get involved?

Want to...

- wrap Histogrammar-Scala as HIVE UDAFs?
- implement cache-aware JIT on the JVM?
- design a modern C++ implementation "the right way?"
- integrate Histogrammar into R's data.frames or ggplot2?
- implement Histogrammar in Javascript for a d3 front-end, thereby connecting Spark or GPUs to the web?

Want to get involved?

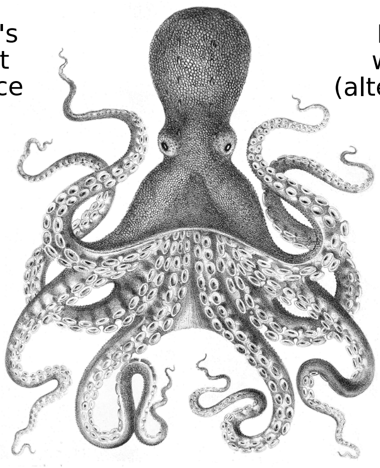
MongoDB's
Javascript
map-reduce

TensorFlow,
Keras, Theano

UNIX command
line tool:

`awk data | hg`

Streaming "tap"
(Storm, S4, etc.)



Binary format
with a schema
(alternative to JSON)

plot.ly, Jupyter
notebooks, etc.

continuously
filling animation
as data arrive

SAGE, MATLAB,
Mathematica, etc.

More back-end ideas...

More front-end ideas...

Want to get involved?

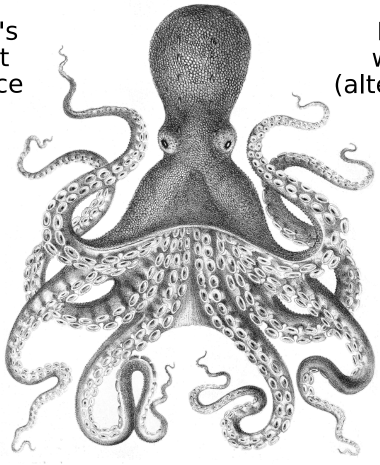
Contact me at **jpivarski@gmail.com**, in Histogrammar's GitHub, or in the conference app!

MongoDB's Javascript map-reduce

TensorFlow, Keras, Theano

UNIX command line tool:
`awk data | hg`

Streaming "tap" (Storm, S4, etc.)



Binary format with a schema (alternative to JSON)

plot.ly, Jupyter notebooks, etc.

continuously filling animation as data arrive

SAGE, MATLAB, Mathematica, etc.

More back-end ideas...

More front-end ideas...