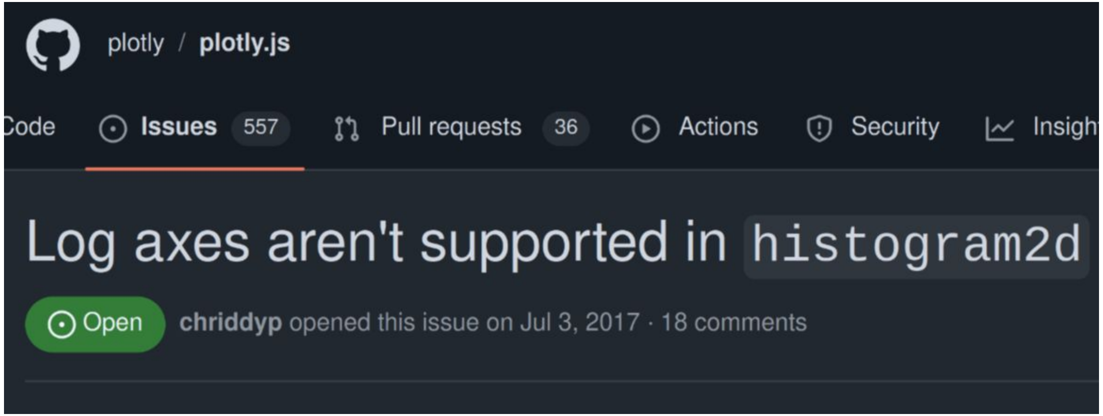
FHist.jl (v0.11) - Histogram for HEP

JuliaHEP 2024 @CERN

Jerry Ling (Harvard University)

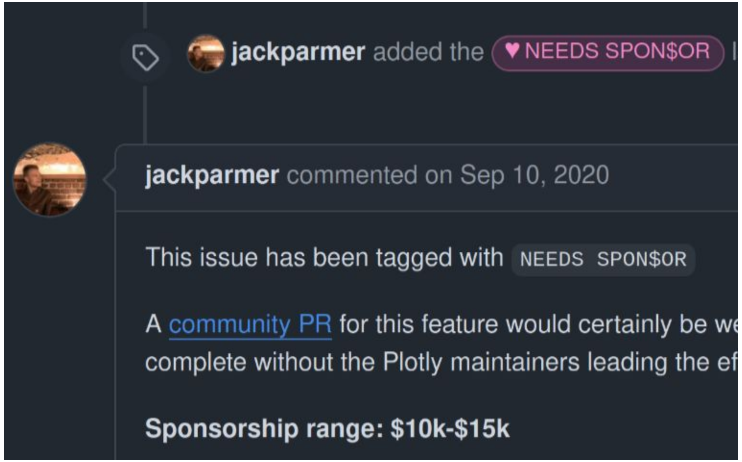
“Why did you make a histogram package?”

- ❖ You may not believe it, but histogram is hard.
- ❖ How hard? Apparently it costs **$10k - $15k** just to support log-scale 2D histogram!



Log axes aren't supported in `histogram2d`

chriddyp opened this issue on Jul 3, 2017 · 18 comments



jackparmer added the ♥NEEDS SPON$OR l

jackparmer commented on Sep 10, 2020

This issue has been tagged with `NEEDS SPON$OR`

A community PR for this feature would certainly be we
complete without the Plotly maintainers leading the ef

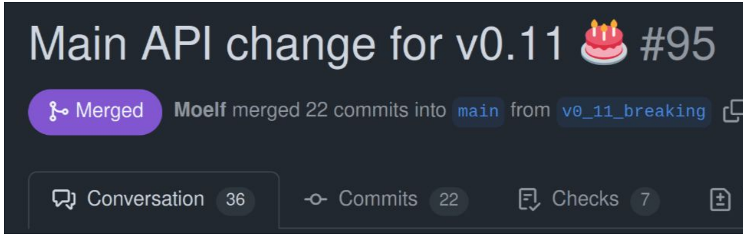
Sponsorship range: $10k-$15k

Outline

Joke aside, histogram package should “just work” and you shouldn’t have to worry about it!

- ❖ Design, features, and performance of FHist.jl
- ❖ Visualization support
- ❖ Potential nice-to-have upgrades

Design - “just work” type signature



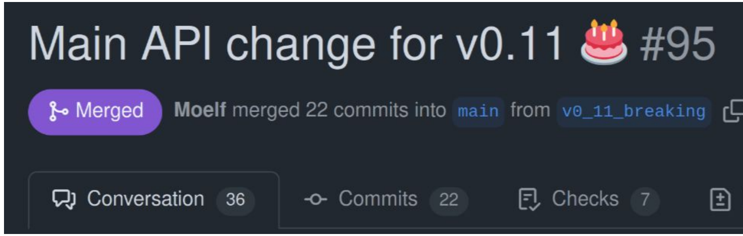
Main API change for v0.11 🎂 #95

Moelf merged 22 commits into main from v0_11_breaking

In the final push for v0.11, Pere convinced me to remove the type parametrization over `binedges` from histograms, so users can easily put histograms into their data types.

Earlier we released on type such as `UnitRange` to dispatch O(1) bin location lookup.

Design - “just work” type signature

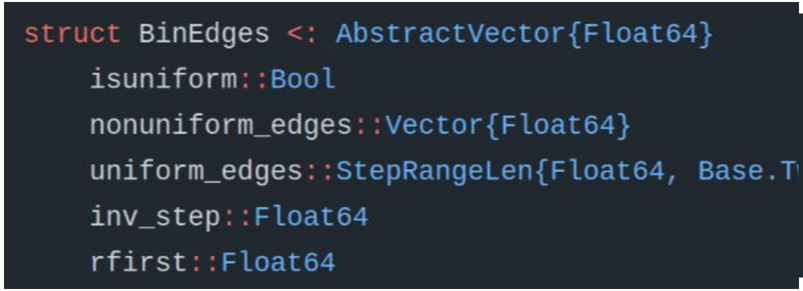


Main API change for v0.11 🎂 #95

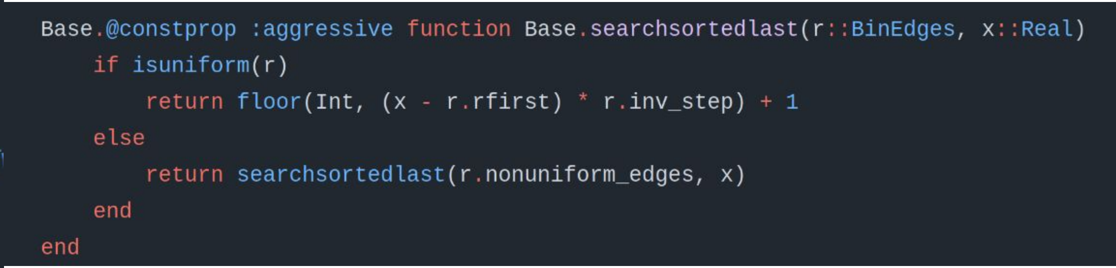
Merged Moelf merged 22 commits into main from v0_11_breaking

Conversation 36 Commits 22 Checks 7

Solution: make a dual-use `BinEdges` type:



```
struct BinEdges <: AbstractVector{Float64}
    isuniform::Bool
    nonuniform_edges::Vector{Float64}
    uniform_edges::StepRangeLen{Float64, Base.T
    inv_step::Float64
    rfirst::Float64
```



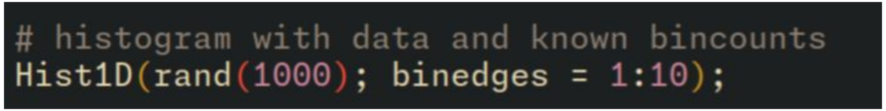
```
Base.@constprop :aggressive function Base.searchsortedlast(r::BinEdges, x::Real)
    if isuniform(r)
        return floor(Int, (x - r.rfirst) * r.inv_step) + 1
    else
        return searchsortedlast(r.nonuniform_edges, x)
    end
end
```

It always records two possible binedges, the `isuniform` jump sometimes are constant-propagated away!

Design - “just work” constructors

Two equally common usage:

- Make a histogram with data already in array
- Make a histogram with known “binedges” or even “bincounts”



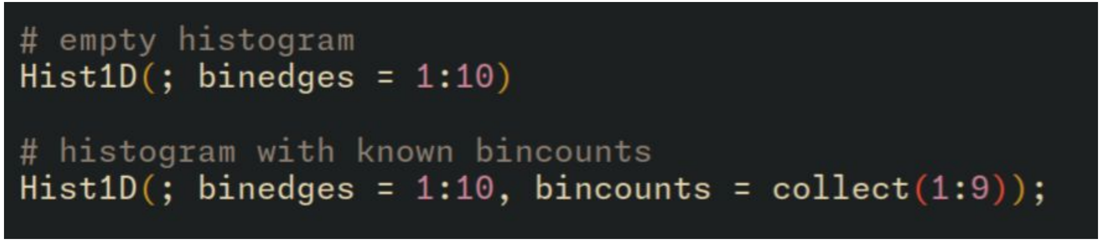
```
# histogram with data and known bincounts
Hist1D(rand(1000); binedges = 1:10);
```

Rule: Data is passed in via positional argument

Design - “just work” constructors

Two equally common usage:

- Make a histogram with data already in array
- Make a histogram with known “binedges” or even “bincounts”



```
# empty histogram
Hist1D(; binedges = 1:10)

# histogram with known bincounts
Hist1D(; binedges = 1:10, bincounts = collect(1:9));
```

Rule: Other attributes are passed in via keyword argument

Features

Unsurprisingly, we try to reference features from ROOT’s TH* class:

- integral
- project
- restrict
- lookup
- normalize
- etc.

Performance

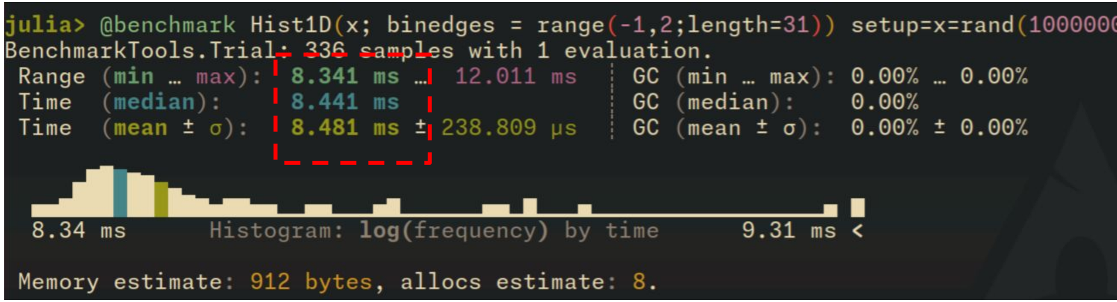
“... you have these random people in Julia that, for some reason, care a lot about histogram performance…”

—Dr. Chris Rackauckas @ JuliaHEP 2023

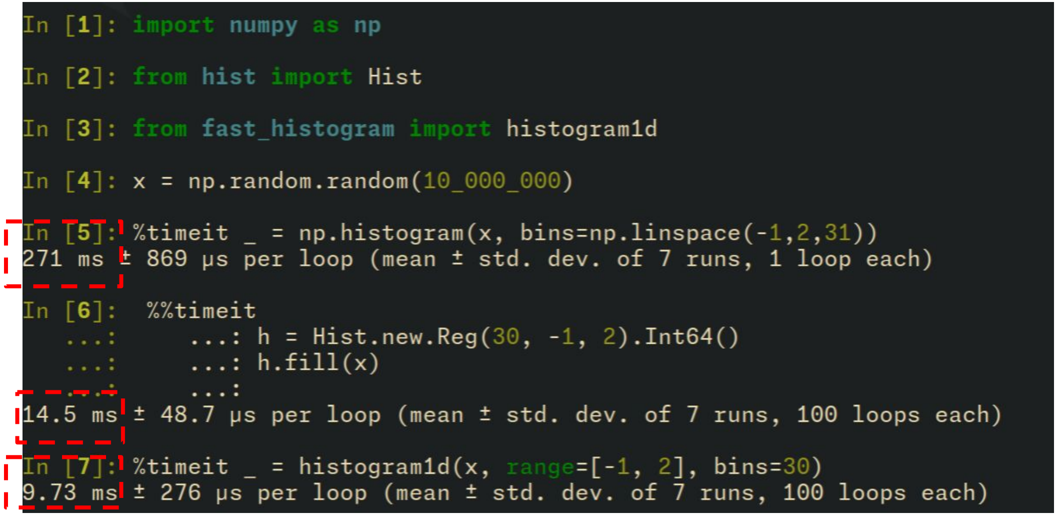
Performance (#87)

We try not to be slower than

C/C++ implementations, we try!



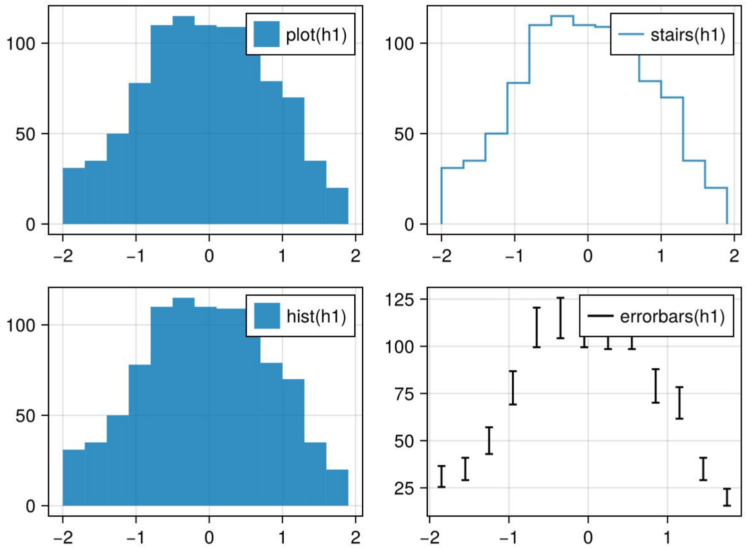
```
julia> @benchmark Hist1D(x; binedges = range(-1,2;length=31)) setup=x=rand(1000000
BenchmarkTools.Trial: 336 samples with 1 evaluation.
 Range (min … max):  8.341 ms … 12.011 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     8.441 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   8.481 ms ± 238.809 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 8.34 ms        Histogram: log(frequency) by time       9.31 ms <

 Memory estimate: 912 bytes, allocs estimate: 8.
```

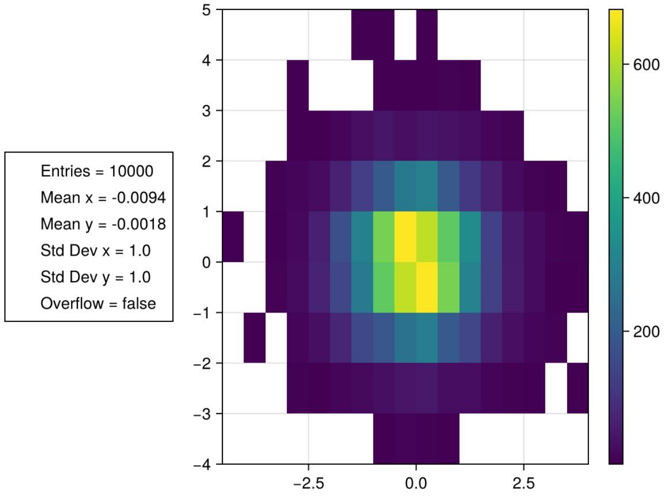


```
In [1]: import numpy as np

In [2]: from hist import Hist

In [3]: from fast_histogram import histogram1d

In [4]: x = np.random.random(10_000_000)

In [5]: %timeit _ = np.histogram(x, bins=np.linspace(-1,2,31))
271 ms ± 869 μs per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [6]:  %%timeit
   ...:      ...: h = Hist.new.Reg(30, -1, 2).Int64()
   ...:      ...: h.fill(x)
   ...:      ...:
14.5 ms ± 48.7 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [7]: %timeit _ = histogram1d(x, range=[-1, 2], bins=30)
9.73 ms ± 276 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

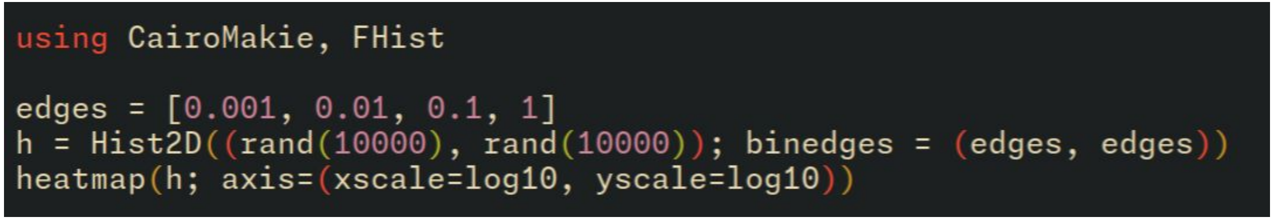
Visualizations - Plots.jl and Makie.jl integration

Many examples for both Plots.jl (special thanks to Prof. Gómez Cadenas) and Makie.jl. Pkg extension mechanism (since Julia v1.9) made life a lot easier.

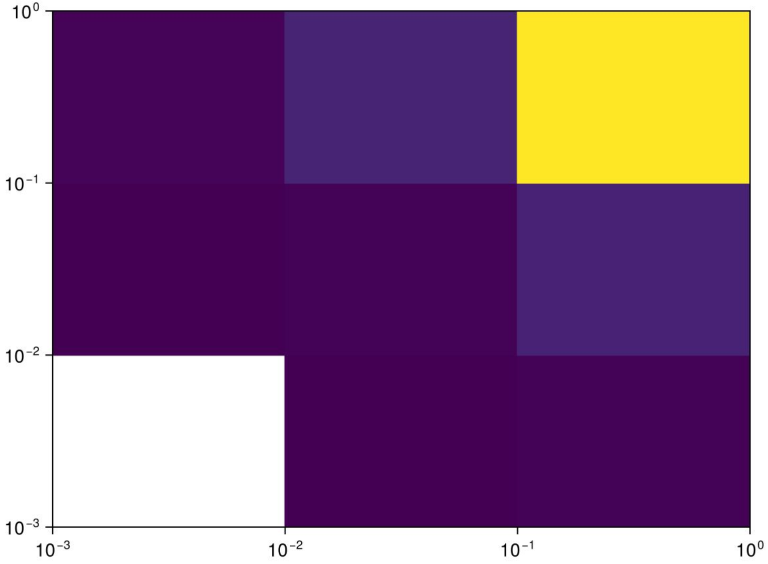




And we can do 2D log plot:



```
using CairoMakie, FHist

edges = [0.001, 0.01, 0.1, 1]
h = Hist2D((rand(10000), rand(10000)); binedges = (edges, edges))
heatmap(h; axis=(xscale=log10, yscale=log10))
```



What upgrades do you want? 🚧

- Categorical (“string”) axis – useful for cutflows
- Alternative value/weight filling – useful for tracking systematics variations
- Serialization format – recently learned CMS W-mass measurement involved a 30 GB C++ Boost-histogram, somehow, you want to save that to disk!
- Integration with statistical frameworks (HS3.jl?)
- GPU-backend?

Let me know what’s the most pressing need!