BAT-2 Status

Oliver Schulz
oschulz@mpp.mpg.de

MAX-PLANCK-GESELLSCHAFT

Max-Planck-Institut für Physik
(Werner-Heisenberg-Institut)

BAT
BAYESIAN ANALYSIS TOOLKIT

BAT Meeting, MPP, Nov. 12th 2018

Ported BAT.jl to Julia v1.0

- BAT.jl initially developed using Julia v0.6
- Julia v1.0 on August 8th 2018
- Julia v1.0 brought many exciting improvements, at the cost of several breaking changes from v0.6
- Had to port our packages ElasticArrays, UnsafeArrays (now registered), MultithreadingTools (now called ParallelProcessingTools) first.
- Hit a non-critical but tricky issue while porting BAT, so took longer than expected. Now all done.

New algorithms for BAT.jl

Currently implementing

- Multi-proposal Metropolis-Hastings: Lolian (see talk later on)
- HMC: Marco (see talk later on)

Future

- Many attractive algorithms to choose from:
  Diffusive Nested Sampling, PolyChord, ...

Doing a fit

Let's fit a peak in a histogram. First, define the likelihood:

In [25]:

```
struct HistFitDensity{F<:Function, P<:ParamShapes, H<:StatsBase.Histogram} <: Abstra
ctDensity
    func::F
    parshapes::P
    hist::H
end

BAT.nparams(hfd::HistFitDensity) = dof(hfd.parshapes)

function BAT.unsafe_density_logval(hfd::HistFitDensity, params::AbstractVector{Float
64}, exec_context::ExecContext)
    p = hfd.parshapes(params)
    bins = hfd.hist.edges[1]
    counts = hfd.hist.weights
    bin_centers = (bins[1:end-1] + bins[2:end]) / 2
    bin_widths = bins[2:end] - bins[1:end-1]
    λ(x, bw) = bw * hfd.func(x, p)
    bin_ll(x, bw, k) = logpdf(Poisson(λ(x, bw)), k)
    sum(broadcast(bin_ll, bin_centers, bin_widths, counts))
end

BAT.exec_capabilities(::typeof(BAT.unsafe_density_logval), hfd::HistFitDensity, para
ms::AbstractVector{<:Real}) =
    ExecCapabilities(0, true, 0, true)
```

Define some ground truth and generate some data

In [49]:

```
truth = (a = 1000, μ = 1.0, σ = 0.5)

data = rand(Normal(truth.μ, truth.σ), truth.a)
hist = StatsBase.fit(StatsBase.Histogram, data, -2:0.1:4, closed = :left)

singlepeak(x::Real, p::NamedTuple) = p.a * pdf(Normal(p.μ, p.σ), x)

Plots.plot(normalize(hist, mode=:density), st = :steps, label = "data", lw=3)
Plots.plot!(-2:0.01:4, x -> singlepeak(x, truth), lw = 2, label = "singlepeak(truth)")
```

Out[49]:

Defining parameters and prior

We'll use a flat prior. Distribution based convenience prior not implemented yet, but we're very close to having them.

In [27]:

```
prior = (
    a = 0.0..10.0^4,
    μ = 0.7..1.3,
    σ = 0.3..0.7
)
```

Out[27]:

```
(a = 0.0..10000.0, μ = 0.7..1.3, σ = 0.3..0.7)
```

All parameters are scalars, how are scalar and array sizes represented in Julia?

In [28]:

```
size(42), size([42]), size([42 42; 42 42])
```

Out[28]:

```
((), (1,), (2, 2))
```

So, in our case, let's make a named tuple with the sizes:

In [29]:

```
param_sizes = map(x -> (), prior)
```

The magic behind named parameters is powered by ParamShapes:

In [30]:
```
params = ParamShapes(param_sizes)
```

Out[30]:
```
ParamShapes{(:a, :μ, :σ),NamedTuple{(:a, :μ, :σ),Tuple{ParameterShapes.ParamDataAccessor{0},ParameterShapes.ParamDataAccessor{0},ParameterShapes.ParamDataAccessor{0}}}}((a = ParameterShapes.ParamDataAccessor{0}((), 0, 1), μ = ParameterShapes.ParamDataAccessor{0}((), 1, 1), σ = ParameterShapes.ParamDataAccessor{0}((), 2, 1)), 3)
```

In [31]:
```
algorithm = MetropolisHastings(MHAccRejProbWeights{Float64}())
density = HistFitDensity(singlepeak, params, hist)
bounds = HyperRectBounds([values(prior)...], reflective_bounds)
```

Out[31]:
```
HyperRectBounds{Float64}(HyperRectVolume{Float64}([0.0, 0.7, 0.3], [10000.0, 1.3, 0.7]), BoundsType[reflective_bounds, reflective_bounds, reflective_bounds])
```

Let's run some MCMC chains:

In [32]:
```
chainspec = MCMCSpec(algorithm, BayesianModel(density, bounds))
samples, sampleids, stats = Base.rand(chainspec, 500, 4)
```

```
INFO (1, 1): Trying to generate 4 viable MCMC chain(s).
DEBUG (1, 1): Generating 32 MCMC chain(s).
DEBUG (1, 1): Testing 32 MCMC chain(s).
DEBUG (1, 1): Found 31 viable MCMC chain(s).
DEBUG (1, 1): Found 29 MCMC chain(s) with at least 4 samples.
DEBUG (1, 1): Generating 32 additional MCMC chain(s).
DEBUG (1, 1): Testing 32 MCMC chain(s).
DEBUG (1, 1): Found 32 viable MCMC chain(s).
DEBUG (1, 1): Found 30 MCMC chain(s) with at least 5 samples.
INFO (1, 1): Selected 4 MCMC chain(s).
INFO (1, 1): Begin tuning of 4 MCMC chain(s).
DEBUG (1, 1): MCMC Tuning cycle 1 finished, 4 chains, 0 tuned, 0 converged.
DEBUG (1, 1): MCMC Tuning cycle 2 finished, 4 chains, 2 tuned, 4 converged.
DEBUG (1, 1): MCMC Tuning cycle 3 finished, 4 chains, 4 tuned, 4 converged.
INFO (1, 1): MCMC tuning of 4 chains successful after 3 cycle(s).
INFO (1, 1): Starting iteration over 4 MCMC chain(s).
DEBUG (1, 1): Starting iteration over MCMC chain 4
DEBUG (1, 1): Starting iteration over MCMC chain 17
DEBUG (1, 1): Starting iteration over MCMC chain 60
DEBUG (1, 1): Starting iteration over MCMC chain 63
```

Out[32]:
```
(DensitySample{Float64,Float64,Float64,Array{Float64,1}}[DensitySample{Float64,
Float64,Float64,SubArray{Float64,1,ElasticArray{Float64,2,1},Tuple{Slice{OneTo
{Int64}},Int64},true}}([1018.65, 0.978264, 0.511801], -89.4364, 0.0, 9.36089),
DensitySample{Float64,Float64,Float64,SubArray{Float64,1,ElasticArray{Float64,
2,1},Tuple{Slice{OneTo{Int64}},Int64},true}}([1024.43, 0.906996, 0.463112], -11
2.464, 0.0, 9.98313e-11), DensitySample{Float64,Float64,Float64,SubArray{Float6
```

MCMC chain output

In [52]:

```
samples.params
```

Out[52]:

```
8694-element ArraysOfArrays.ArrayOfSimilarArrays{Float64,1,1,2,ElasticArrays.El
asticArray{Float64,2,1}}:
 [516.839, 994.889, -0.972503, 2.01085, 0.491971]
 [501.119, 1010.28, -0.93772, 2.00388, 0.500963]
 [510.552, 1080.68, -0.974919, 1.96444, 0.491421]
 [602.925, 926.906, -0.933642, 2.02725, 0.512487]
 [647.246, 912.904, -1.2065, 1.9539, 0.490457]
 [514.347, 1039.56, -1.00464, 2.01787, 0.490577]
 [535.632, 1239.07, -0.8771, 1.86634, 0.499658]
 [564.609, 953.01, -1.00822, 2.02643, 0.489923]
 [409.838, 733.845, -0.876213, 2.01714, 0.495389]
 [487.963, 978.071, -0.95187, 1.99432, 0.483955]
 [439.52, 1037.51, -0.952744, 1.96539, 0.481321]
 [472.869, 1027.88, -0.955272, 1.97292, 0.484265]
 [542.031, 972.549, -0.932392, 2.0141, 0.495034]
 ⋮
 [81.2725, 756.996, -0.689688, 1.91547, 0.397041]
 [624.725, 1187.01, -1.12496, 2.04111, 0.520028]
 [447.386, 1061.07, -1.01649, 1.99066, 0.492662]
 [513.791, 1081.19, -1.01746, 2.01174, 0.496958]
 [511.401, 957.152, -1.08289, 2.00873, 0.487197]
 [156.279, 812.796, -1.053, 2.03088, 0.521435]
 [1193.69, 1602.23, -1.32726, 2.15377, 0.633281]
 [466.242, 887.184, -0.952314, 1.9826, 0.553747]
 [898.052, 1657.54, -0.888646, 2.10262, 0.49344]
 [542.411, 1096.94, -1.02551, 2.00319, 0.500919]
 [550.642, 1134.55, -1.00681, 2.02791, 0.496638]
 [527.794, 1115.82, -1.00533, 2.00651, 0.49367]
```

Let's make use of our parameter shapes

In [34]:

```
params(samples.params)
```

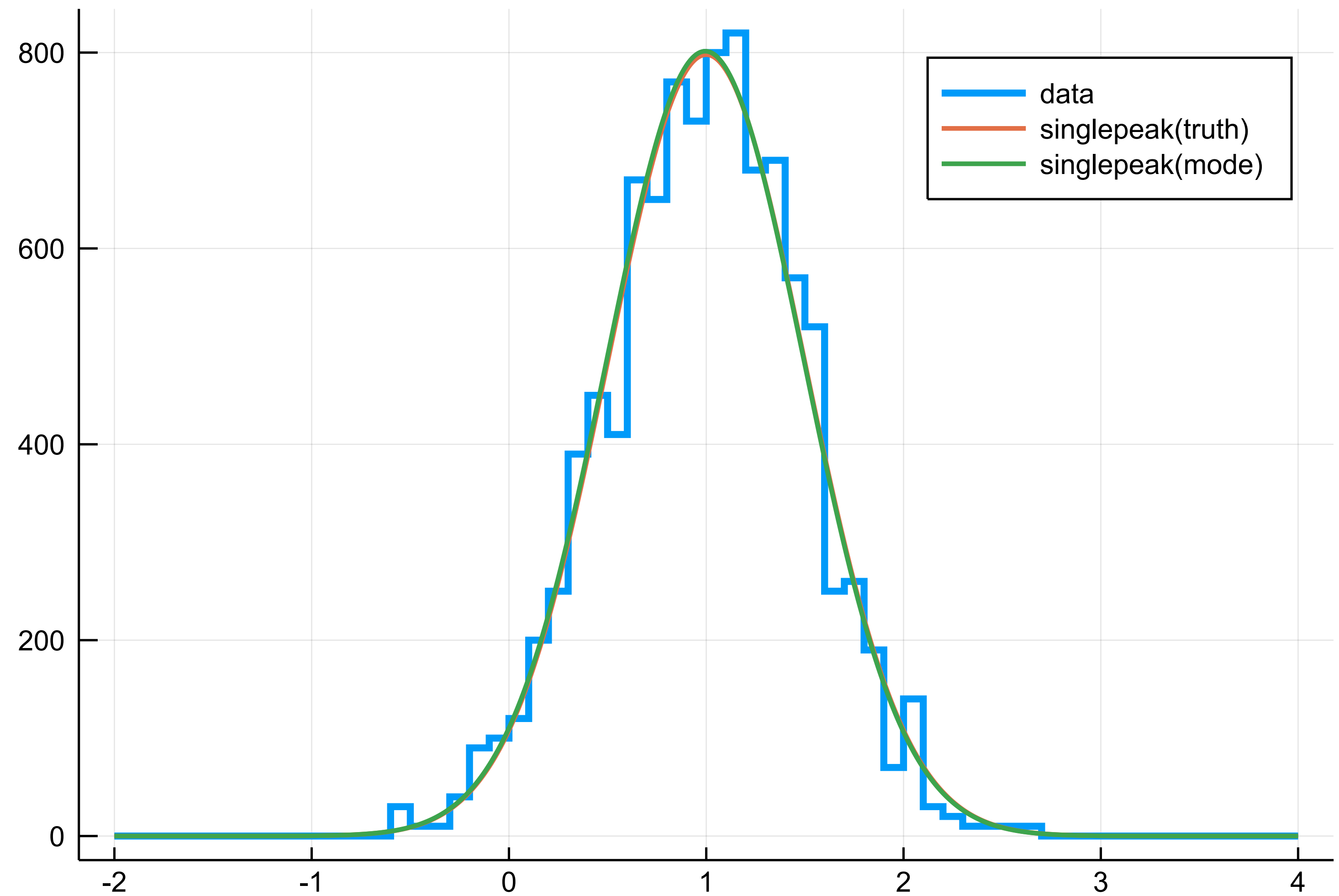
Out[34]:

```
Table with 3 columns and 9610 rows:
      a        μ         σ
    ┌───────────────────────────
 1  │ 1018.65  0.978264  0.511801
 2  │ 1024.43  0.906996  0.463112
 3  │ 1257.95  0.99375   0.380503
 4  │ 1168.15  0.977814  0.517565
 5  │ 1077.73  0.95874   0.507114
 6  │ 993.088  0.90032   0.472732
 7  │ 955.573  0.989776  0.513172
 8  │ 1066.58  0.940317  0.471872
 9  │ 2050.41  0.862348  0.622803
 10 │ 1009.63  0.968416  0.502858
 11 │ 1043.83  1.0154    0.51248
 12 │ 1062.81  0.992361  0.528249
 13 │ 1142.71  1.04048   0.484192
 14 │ 1042.49  0.998841  0.526335
 15 │ 1080.18  1.06297   0.603773
 16 │ 1002.36  1.00036   0.531414
 17 │ 922.514  1.01021   0.530355
 18 │ 1138.95  1.07324   0.463134
 19 │ 307.124  0.964637  0.334771
 20 │ 1033.77  0.99638   0.497028
 21 │ 987.21   0.978166  0.495876
 22 │ 1031.84  0.949272  0.521529
 23 │ 1007.66  0.963333  0.486618
 ⋮  │    ⋮        ⋮         ⋮
```

In [36]:

```
Plots.plot(normalize(hist, mode=:density), st = :steps, label = "data", lw=3)
Plots.plot!(-2:0.01:4, x -> singlepeak(x, truth), lw = 2, label = "singlepeak(trut
h)")
Plots.plot!(-2:0.01:4, x -> singlepeak(x, params(stats.mode)), lw = 2, label = "sing
lepeak(mode)")
```

Out[36]:



In [37]:
```
println("Truth: $truth")
println("Mode: $(params(stats.mode))")
println("Mean: $(params(stats.param_stats.mean))")
println("Covariance: $(stats.param_stats.cov)")
```

```
Truth: (a = 1000, μ = 1.0, σ = 0.5)
Mode: (a = 1003.8064255234221, μ = 0.9952162348141104, σ = 0.49970511054314976)
Mean: (a = 1001.2127612521659, μ = 0.9932832405520088, σ = 0.5016018905791119)
Covariance: [973.532 -0.018659 0.00328854; -0.018659 0.000247819 4.2739e-6; 0.0
0328854 4.2739e-6 0.000124819]
```

Will it compose with Measurements.jl?

In [38]:
```
pfit = params(
    measurement.(
        stats.param_stats.mean,
        .√(diag(stats.param_stats.cov))
    )
)
```

Out[38]:
```
(a = 1001.2127612521659 ± 31.201467045926655, μ = 0.9932832405520088 ± 0.015742
266006975585, σ = 0.5016018905791119 ± 0.011172224844601944)
```

In [39]:
```
println("Relative peak resolution: $(pfit.σ / pfit.μ)")
```

```
Relative peak resolution: 0.5049938125406715 ± 0.013804654728826462
```

Now let's fit two peaks

In [55]:

```
truth = (a = [500, 1000], μ = [-1.0, 2.0], σ = 0.5)
dist = Normal.(truth.μ, truth.σ)
data = vcat(rand.(dist, truth.a)...)

hist = StatsBase.fit(StatsBase.Histogram, data, -2:0.1:4, closed = :left)

multipeak(x::Real, p::NamedTuple) = sum(p.a .* pdf.(Normal.(p.μ, p.σ), x))
```

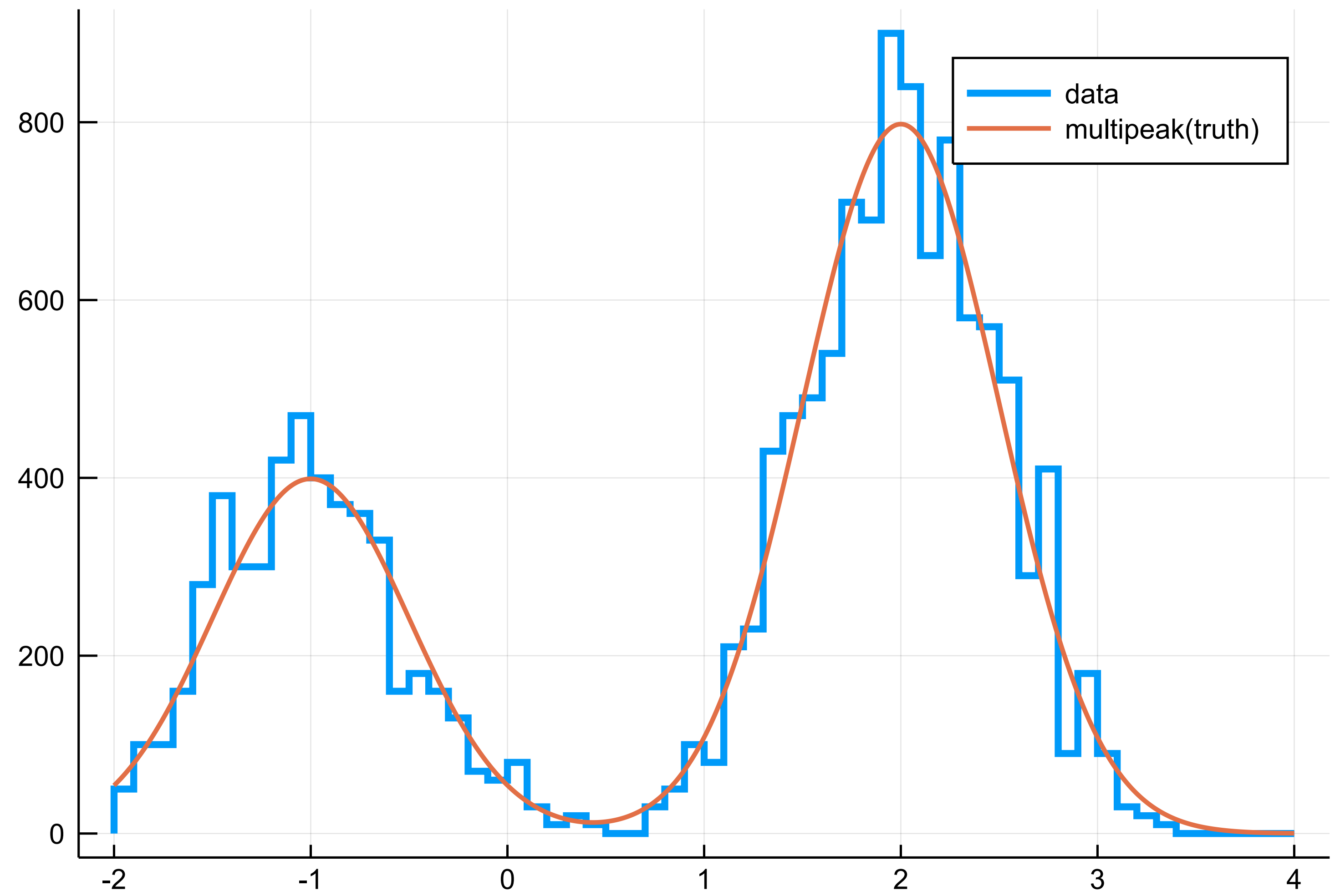
Out[55]:

```
multipeak (generic function with 1 method)
```

In [56]:

```
Plots.plot(normalize(hist, mode=:density), st = :steps, label = "data", lw=3)
Plots.plot!(-2:0.01:4, x -> multipeak(x, truth), lw = 2, label = "multipeak(truth)")
```

Out[56]:



In [42]:
```
prior = (
    a = [0.0..10.0^4, 0.0..10.0^4],
    μ = [-2.0..0.0, 1.0..3.0],
    σ = Scalar(0.3..0.7)
)
```

Out[42]:
```
(a = Interval{:closed,:closed,Float64}[0.0..10000.0, 0.0..10000.0], μ = Interva
l{:closed,:closed,Float64}[-2.0..0.0, 1.0..3.0], σ = 0.3..0.7)
```

In [57]:
```
map(size, prior)
```

Out[57]:
```
(a = (2,), μ = (2,), σ = ())
```

In [43]:

```
params = ParamShapes(map(size, prior))

density = HistFitDensity(multipeak, params, hist)
bounds = HyperRectBounds(vcat(map(x -> x[:], values(prior))...), reflective_bounds)

chainspec = MCMCSpec(algorithm, BayesianModel(density, bounds))

samples, sampleids, stats = Base.rand(chainspec, 500, 4)
```

```
INFO (1, 1): Trying to generate 4 viable MCMC chain(s).
DEBUG (1, 1): Generating 32 MCMC chain(s).
DEBUG (1, 1): Testing 32 MCMC chain(s).
DEBUG (1, 1): Found 32 viable MCMC chain(s).
DEBUG (1, 1): Found 28 MCMC chain(s) with at least 5 samples.
DEBUG (1, 1): Generating 32 additional MCMC chain(s).
DEBUG (1, 1): Testing 32 MCMC chain(s).
DEBUG (1, 1): Found 30 viable MCMC chain(s).
DEBUG (1, 1): Found 22 MCMC chain(s) with at least 6 samples.
INFO (1, 1): Selected 4 MCMC chain(s).
INFO (1, 1): Begin tuning of 4 MCMC chain(s).
DEBUG (1, 1): MCMC Tuning cycle 1 finished, 4 chains, 0 tuned, 0 converged.
DEBUG (1, 1): MCMC Tuning cycle 2 finished, 4 chains, 0 tuned, 4 converged.
DEBUG (1, 1): MCMC Tuning cycle 3 finished, 4 chains, 0 tuned, 4 converged.
DEBUG (1, 1): MCMC Tuning cycle 4 finished, 4 chains, 0 tuned, 4 converged.
DEBUG (1, 1): MCMC Tuning cycle 5 finished, 4 chains, 0 tuned, 4 converged.
DEBUG (1, 1): MCMC Tuning cycle 6 finished, 4 chains, 1 tuned, 4 converged.
DEBUG (1, 1): MCMC Tuning cycle 7 finished, 4 chains, 4 tuned, 4 converged.
INFO (1, 1): MCMC tuning of 4 chains successful after 7 cycle(s).
INFO (1, 1): Starting iteration over 4 MCMC chain(s).
DEBUG (1, 1): Starting iteration over MCMC chain 6
DEBUG (1, 1): Starting iteration over MCMC chain 41
DEBUG (1, 1): Starting iteration over MCMC chain 47
DEBUG (1, 1): Starting iteration over MCMC chain 55
```

Out[43]:

```
(DensitySample{Float64,Float64,Float64,Array{Float64,1}}[DensitySample{Float64,
```

In [45]:
```
parsamples = params(samples.params)
```

Out[45]:

Table with 3 columns and 8694 rows:

	a	μ	σ
1	[516.839, 994.889]	[-0.972503, 2.01085]	0.491971
2	[501.119, 1010.28]	[-0.93772, 2.00388]	0.500963
3	[510.552, 1080.68]	[-0.974919, 1.96444]	0.491421
4	[602.925, 926.906]	[-0.933642, 2.02725]	0.512487
5	[647.246, 912.904]	[-1.2065, 1.9539]	0.490457
6	[514.347, 1039.56]	[-1.00464, 2.01787]	0.490577
7	[535.632, 1239.07]	[-0.8771, 1.86634]	0.499658
8	[564.609, 953.01]	[-1.00822, 2.02643]	0.489923
9	[409.838, 733.845]	[-0.876213, 2.01714]	0.495389
10	[487.963, 978.071]	[-0.95187, 1.99432]	0.483955
11	[439.52, 1037.51]	[-0.952744, 1.96539]	0.481321
12	[472.869, 1027.88]	[-0.955272, 1.97292]	0.484265
13	[542.031, 972.549]	[-0.932392, 2.0141]	0.495034
14	[199.987, 823.624]	[-0.845006, 2.10596]	0.471728
15	[366.274, 999.928]	[-0.931205, 1.8674]	0.461839
16	[507.992, 1014.12]	[-0.97373, 2.00023]	0.48841
17	[403.656, 820.849]	[-0.891321, 1.96158]	0.469181
18	[474.318, 1106.48]	[-0.97337, 2.00622]	0.48235
19	[644.497, 975.223]	[-1.06911, 2.10065]	0.487928
20	[516.165, 995.489]	[-0.959763, 1.99936]	0.479889
21	[447.704, 989.541]	[-0.937484, 1.97613]	0.48883
22	[570.17, 956.429]	[-0.932757, 1.9775]	0.506684
23	[486.0, 1080.19]	[-0.885937, 1.94579]	0.497844
⋮	⋮	⋮	⋮

In [46]:

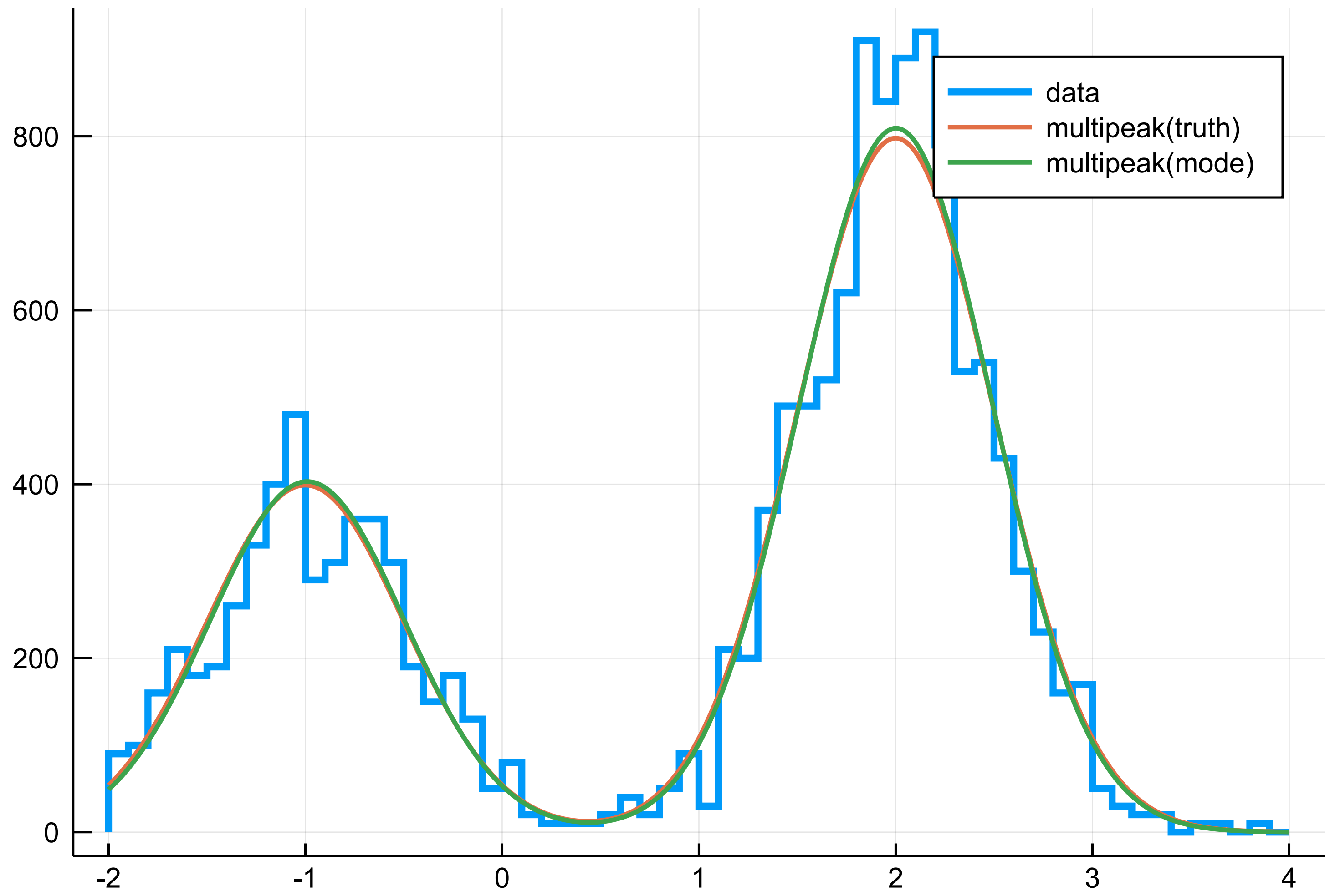
```
println("Truth: $truth")
println("Mode: $(params(stats.mode))")
println("Mean: $(params(stats.param_stats.mean))")
println("Covariance: $(stats.param_stats.cov)")
```

```
Truth: (a = [500, 1000], μ = [-1.0, 2.0], σ = 0.5)
Mode: (a = [496.967, 997.955], μ = [-0.990601, 2.00197], σ = 0.4919085363809150
3)
Mean: (a = [495.086, 1002.59], μ = [-0.983993, 2.00399], σ = 0.490470525619341
8)
Covariance: [500.2 -8.53296 0.00240438 0.0021036 0.00411009; -8.53296 1095.08 -
0.0640278 0.0232947 0.00709811; 0.00240438 -0.0640278 0.000547686 2.52479e-6 -
1.96441e-5; 0.0021036 0.0232947 2.52479e-6 0.000221922 1.21794e-6; 0.00411009
0.00709811 -1.96441e-5 1.21794e-6 9.18994e-5]
```

In [47]:

```
Plots.plot(normalize(hist, mode=:density), st = :steps, label = "data", lw=3)
Plots.plot!(-2:0.01:4, x -> multipeak(x, truth), lw = 2, label = "multipeak(truth)")
Plots.plot!(-2:0.01:4, x -> multipeak(x, params(stats.mode)), lw = 2, label = "multi
peak(mode)")
```

Out[47]:



Summary and Outlook

- BAT now runs on long-term stable Julia v1.0 - smoother sailing in the future
- Completed major code reorganization to ease cooperation
- Rewrote internals of MH sampler, much cleaner now
- New types like `BayesianModel`
- Named parameters via new ParameterShapes.jl
- To do:
  - New sampling algorithms
  - More unit and performance tests
  - More plotting recipes
  - Models for frequent use case
  - Connect to likelihoods in external processes via pipes