Gaussian Mixture

```
#    margin-left: 1%;
#    margin-right: 5%;
html"""<style>
main {
    margin: 0 auto;
    max-width: 90%;
    padding-left: max(50px, 1%);
    padding-right: max(253px, 10%);
    # 253px to accomodate TableOfContents(aside=true)
}
"""
```

```
using Pkg, DrWatson, PlutoUI
```

Table of Contents

Gaussian Mixture


```
begin
  PlutoUI.TableOfContents()
end
```

```
versioninfo()
```

```
Julia Version 1.10.2
Commit bd47eca2c8a (2024-03-01 10:14 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: 32 × Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-15.0.7 (ORCJIT, haswell)
Threads: 16 default, 0 interactive, 8 GC (on 32 virtua
l cores)
Environment:
  JULIA_PKG_SERVER = https://mirrors.tuna.tsinghua.ed
u.cn/julia
  JULIA_REVISE_WORKER_ONLY = 1
```

1.0 Import all packages

```
TaskLocalRNG()
```

```julia
begin
    using Distributions
    using FillArrays
    using StatsPlots

    using LinearAlgebra
    using Random
    using Turing

    # Set a random seed.
    Random.seed!(3)
end
```

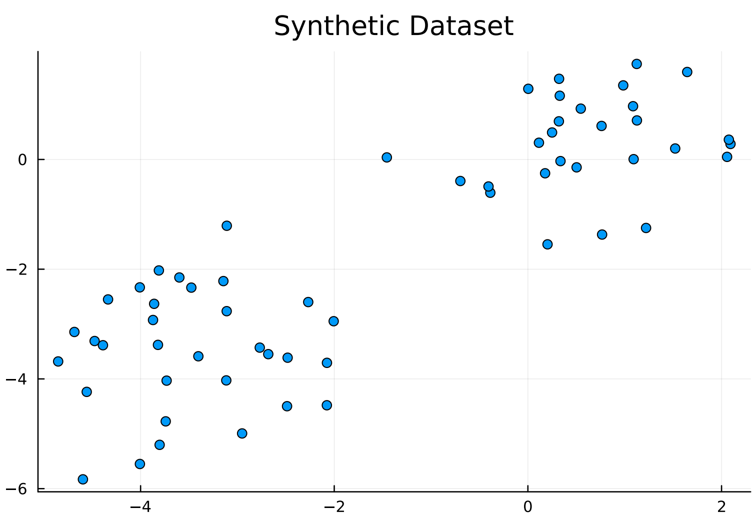
1.1 Simulate some data

```
2×60 Matrix{Float64}:
 -3.80334  -3.87173  2.09236   1.64516  …  -4.68366  0.24971   -3
 -5.20162  -2.9261   0.282404  1.59448     -3.14423  0.491857  -4
```

```julia
begin
    # Define Gaussian mixture model.
    w = [0.5, 0.5]
    μ = [-3.5, 0.5]
    mixturemodel = MixtureModel([MvNormal(Fill(μₖ, 2),
    Distributions.I) for μₖ in μ], w)

    # We draw the data points.
    N = 60
    @time x = rand(mixturemodel, N)
end
```

```
0.573396 seconds (491.87 k allocations: 33.482 MiB, 99.98% compilation time)
```




```julia
scatter(x[1, :], x[2, :]; legend=false, title="Synthetic Dataset")
```

1.2 Establish Turing model

```
gaussian_mixture_model (generic function with 2 methods)
```

```
@model function gaussian_mixture_model(x)
    # Draw the parameters for each of the K=2 clusters from a standard normal distribution.
    K = 2
    μ ~ MvNormal(Zeros(K), I)

    # Draw the weights for the K clusters from a Dirichlet distribution with parameters αₖ = 1.
    w ~ Dirichlet(K, 1.0)
    # Alternatively, one could use a fixed set of weights.
    # w = fill(1/K, K)

    # Construct categorical distribution of assignments.
    distribution_assignments = Categorical(w)

    # Construct multivariate normal distributions of each cluster.
    D, N = size(x)
    distribution_clusters = [MvNormal(Fill(μₖ, D), I) for μₖ in μ]

    # Draw assignments for each datum and generate it from the multivariate normal distribution.
    k = Vector{Int}(undef, N)
    for i in 1:N
        k[i] ~ distribution_assignments
        x[:, i] ~ distribution_clusters[k[i]]
    end

    return k
end
```

1.3 Sampling by PG+HMC

- PG for the discrete K. HMC for continuous μ and w.

	iteration	chain	μ[1]	μ[2]	w[1]	w[2]
1	1	1	-3.17473	-0.338518	0.493137	0.506863
2	2	1	-2.95212	0.358465	0.648418	0.351582
3	3	1	-3.85494	0.536792	0.422491	0.577509
4	4	1	-2.98575	0.291792	0.545014	0.454986
5	5	1	-3.71472	0.548404	0.744618	0.255382
6	6	1	-3.34117	0.374166	0.558974	0.441026
7	7	1	-3.34117	0.374166	0.558974	0.441026
8	8	1	-3.47253	0.562244	0.55249	0.44751
9	9	1	-3.37564	0.294521	0.537982	0.462018
10	10	1	-3.50283	0.48389	0.436915	0.563085

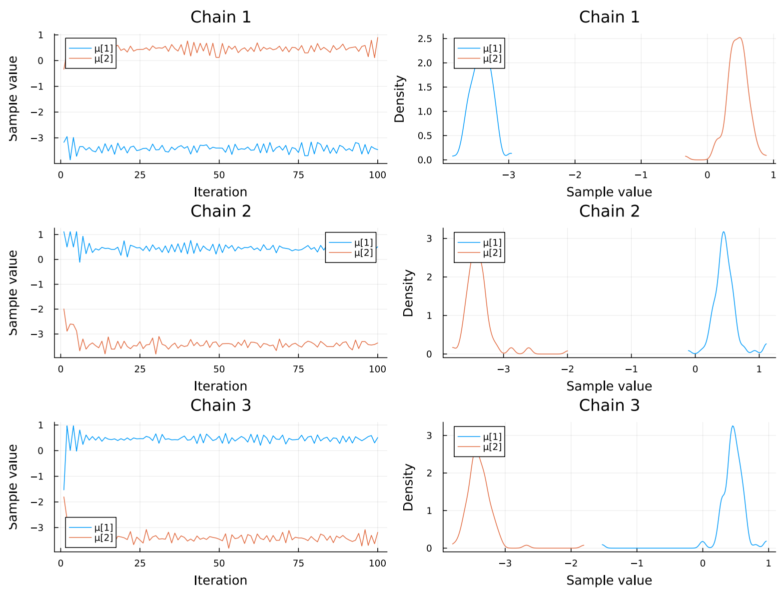
more

```
begin
    model = gaussian_mixture_model(x);
    # k, μ, w are all mapped into the sampler?
    sampler = Gibbs(PG(100, :k), HMC(0.05, 10, :μ, :w))
    nsamples = 100
    nchains = 3
    @time chains = sample(model, sampler, MCMCThreads(),
    nsamples, nchains);
end
```

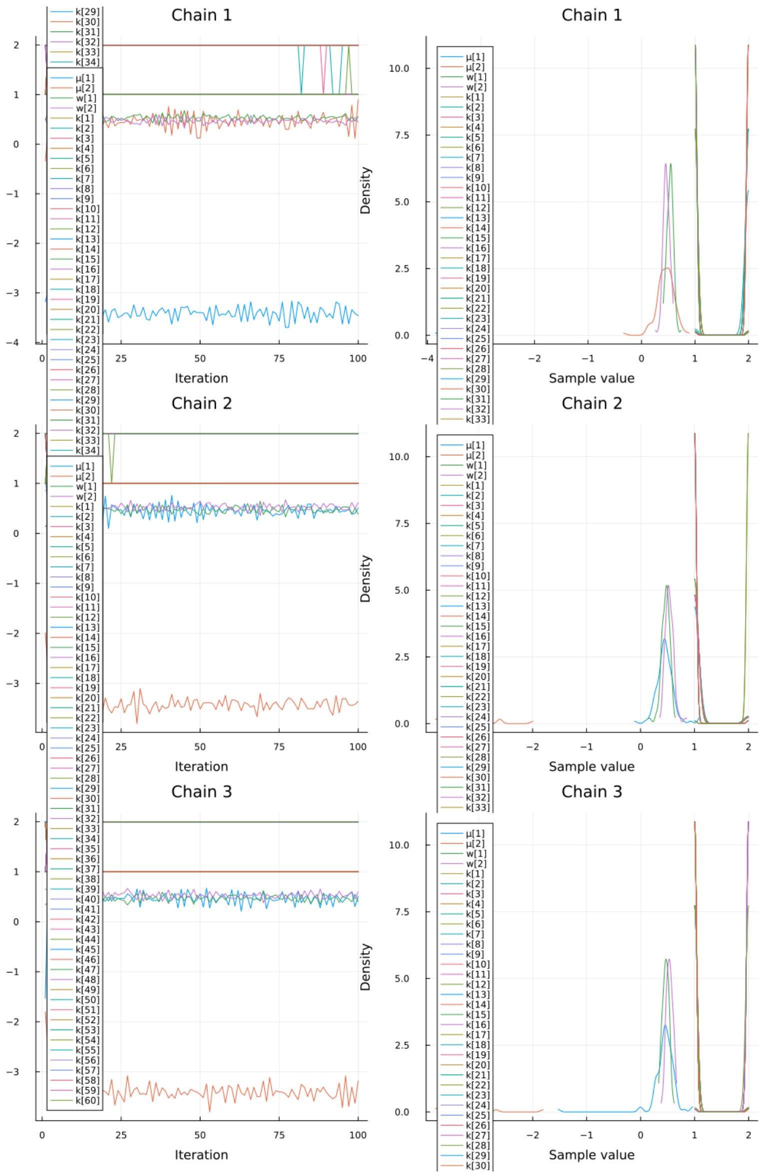
100%

246.782446 seconds (1.30 G allocations: 108.373 GiB, 9.86% gc time, 29.66% compilation time)

```
chains
```




```
plot(chains[["μ[1]", "μ[2]"]]; colordim=:parameter,
legend=true)
```

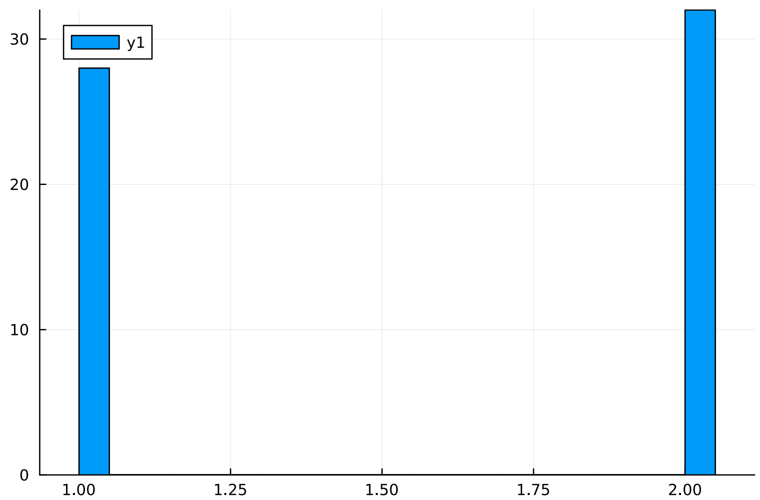



```
plot(chains; colordim=:parameter, legend=true, size=
(1000,1500))
```

```
3-dimensional AxisArray{Float64,3,...} with axes:
    :iter, 1:1:100
    :var, [Symbol("μ[1]"), Symbol("μ[2]"), Symbol("w[1]"), Symb
    :chain, 1:3
And data, a 100×65×3 Array{Float64, 3}:
[:, :, 1] =
 -3.17473  -0.338518  0.493137  0.506863  …  2.0  2.0  1.0  1.0
 -2.95212   0.358465  0.648418  0.351582     1.0  1.0  1.0  2.0
 -3.85494   0.536792  0.422491  0.577509     1.0  1.0  1.0  2.0
 -2.98575   0.291792  0.545014  0.454986     1.0  1.0  1.0  2.0
 -3.71472   0.548404  0.744618  0.255382     1.0  1.0  1.0  2.0
 -3.34117   0.374166  0.558974  0.441026  …  1.0  1.0  1.0  2.0
 -3.34117   0.374166  0.558974  0.441026     1.0  1.0  1.0  2.0
  ⋮                                       ⋱            ⋮
 -3.629     0.550979  0.561428  0.438572     1.0  1.0  1.0  2.0
 -3.25971   0.589129  0.620074  0.379926  …  1.0  1.0  1.0  2.0
 -3.62434   0.153246  0.475385  0.524615     1.0  1.0  1.0  2.0
 -3.33782   0.788136  0.54756   0.45244      1.0  1.0  1.0  2.0
 -3.41603   0.113558  0.463805  0.536195     1.0  1.0  1.0  2.0
 -3.45961   0.897516  0.555307  0.444693     1.0  1.0  1.0  2.0

[:, :, 2] =
  1.11545   -1.99098  0.138071  0.861929  …  2.0  2.0  2.0  2.0
  0.501362  -2.88465  0.183156  0.816844     2.0  2.0  2.0  1.0
  1.11487   -2.5986   0.289975  0.710025     2.0  2.0  2.0  1.0
  0.500535  -2.61454  0.349798  0.650202     2.0  2.0  2.0  1.0
  1.11584   -2.87466  0.285701  0.714299     2.0  2.0  2.0  1.0
 -0.112998  -3.67492  0.574123  0.425877  …  2.0  2.0  2.0  1.0
  0.928263  -3.1998   0.497663  0.502337     2.0  2.0  2.0  1.0
  ⋮                                       ⋱            ⋮
  0.443253  -3.30328  0.52756   0.47244      2.0  2.0  2.0  1.0
  0.443253  -3.30328  0.52756   0.47244   …  2.0  2.0  2.0  1.0
```

```
chains.value
```




```
#last iteration of the 3rd chain of 60 k values (cluster
assignments)
histogram(chains.value[iter=100][5:64,3], bins=20)
```

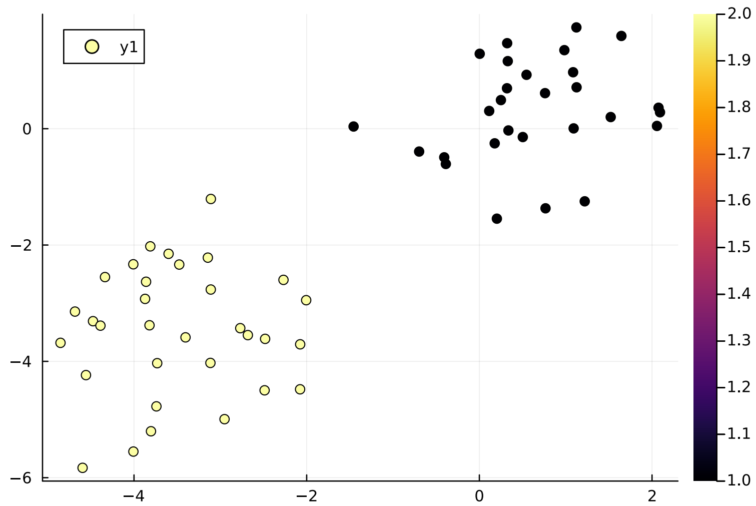
```
(varname_to_symbol = OrderedDict(μ[1] ⇒ Symbol("μ[1]"), μ[2] ⇒
```

```
chains.info
```

[0.460472, 0.531358, 0.533908]

```
begin
    @show median.(eachcol(chains[:"μ[1]"]))
    @show median.(eachcol(chains[:"μ[2]"]))
    @show median.(eachcol(chains[:"w[1]"]))
    @show median.(eachcol(chains[:"w[2]"]))
end
```

```
median.(eachcol(chains[$(QuoteNode("μ[1]"))])) = [-
3.405617489183758, 0.44674296000216607, 0.463459933397
3022]
median.(eachcol(chains[$(QuoteNode("μ[2]"))])) = [0.45
82280003677017, -3.420573823097018, -3.406275476316240
6]
median.(eachcol(chains[$(QuoteNode("w[1]"))])) = [0.53
95282753183839, 0.4686417827884136, 0.466092321742558
7]
median.(eachcol(chains[$(QuoteNode("w[2]"))])) = [0.46
047172468161607, 0.5313582172115864, 0.533907678257441
2]
```




```
begin
    cmap = Dict(1 => :red, 2 => :blue)
    # Color function based on color map
    color_func(value) = cmap[get(cmap, value, :gray)]  #
    Default to gray if not in map
    color_vec = chains.value[iter=100][5:64,3]
    scatter(x[1, :], x[2, :]; marker_z=color_vec) #
    markerfacecolor=map(color_func, color_vec),
    title="Synthetic Dataset")
end
```

	iteration	chain	μ[1]	μ[2]	w[1]	w[2]
1	1	1	-0.917817	0.514631	0.88358	0.11642
2	2	1	-2.23336	1.17691	0.638822	0.361178
3	3	1	-2.73592	0.438788	0.752347	0.247653
4	4	1	-3.77493	0.554984	0.441144	0.558856
5	5	1	-3.0009	0.43676	0.721961	0.278039
6	6	1	-3.61718	0.811352	0.476689	0.523311
7	7	1	-3.4402	0.278046	0.527244	0.472756
8	8	1	-3.43152	0.478669	0.385599	0.614401
9	9	1	-3.45661	0.411029	0.638697	0.361303
10	10	1	-3.27962	0.564529	0.441955	0.558045
	more					

```
begin
@time sample(model, sampler, MCMCThreads(), nsamples, nchains)
end
```

```
100%
```

```
226.930748 seconds (1.27 G allocations: 106.401 GiB, 10.05% gc time)
```

1.4 Sampling by NUTS(). Failed due to ForwardDiff error.

- k is discrete, so NUTS/HMC sampler will fail (no gradient).

```
ArgumentError: invalid index:
Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag,
Float64}}
(2.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0) of
type
ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLT
ag, Float64}, Float64, 11}
```

Stack trace

Here is what happened, the most recent locations are first:

1. **to_index**(i::ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}) @ *indices.jl:300*
2. **to_index**(A::Vector{Distributions.MvNormal{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}, PDMats.ScalMat{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}}, FillArrays.Fill{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}, 1, Tuple{Base.OneTo{Int64}}}}}, i::ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}) @ *indices.jl:277*
3. **_to_indices1**(A::Vector{Distributions.MvNormal{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}, PDMats.ScalMat{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}}, FillArrays.Fill{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}, 1, Tuple{Base.OneTo{Int64}}}}}, inds::Tuple{Base.OneTo{Int64}}, I1::ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}) @ *indices.jl:359*
4. **to_indices** @ *indices.jl:354*
5. **to_indices** @ *indices.jl:345*
6. **getindex** @ *abstractarray.jl:1291*
7. **gaussian_mixture_model**(__model__::DynamicPPL.Model{typeof(Main.var"workspace#4".gaussian_mixture_model), (:x,), (), (), Tuple{Matrix{Float64}}, Tuple{}, DynamicPPL.DefaultContext}, __varinfo__::DynamicPPL.ThreadSafeVarInfo{DynamicPPL.TypedVarInfo{@NamedTuple{μ::DynamicPPL.Metadata{Dict{AbstractPPL.VarName{:μ, Setfield.IdentityLens}, Int64}, Vector{Distributions.ZeroMeanIsoNormal{Tuple{Base.OneTo{Int64}}}}, Vector{AbstractPPL.VarName{:μ, Setfield.IdentityLens}}, Vector{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}}, Vector{Set{DynamicPPL.Selector}}}, w::DynamicPPL.Metadata{Dict{AbstractPPL.VarName{:w, Setfield.IdentityLens}, Int64}, Vector{Distributions.Dirichlet{Float64, FillArrays.Fill{Float64, 1, Tuple{Base.OneTo{Int64}}}, Float64}}, Vector{AbstractPPL.VarName{:w,

```
Setfield.IdentityLens}},
Vector{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}},
Vector{Set{DynamicPPL.Selector}}},
k::DynamicPPL.Metadata{Dict{AbstractPPL.VarName{:k,
Setfield.IndexLens{Tuple{Int64}}}, Int64},
Vector{Distributions.Categorical{Float64,
Vector{Float64}}}, Vector{AbstractPPL.VarName{:k,
Setfield.IndexLens{Tuple{Int64}}}},
Vector{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}},
Vector{Set{DynamicPPL.Selector}}}},
ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}},
Vector{Base.RefValue{ForwardDiff.Dual{ForwardDiff.Tag{DynamicPPL.DynamicPPLTag, Float64}, Float64, 11}}}},
__context__::DynamicPPL.SamplingContext{DynamicPPL.Sampler{Turing.Inference.NUTS{ADTypes.AutoForwardDiff{0, Nothing}, (), AdvancedHMC.DiagEuclideanMetric}},
DynamicPPL.DefaultContext, Random.TaskLocalRNG},
x::Matrix{Float64}) @ Other cell: line 22
```

```
20      for i in 1:N
21          k[i] ~ distribution_assignments
22          x[:, i] ~ distribution_clusters[k[i]]
23      end
24
```

cell preview

8. Show more...

```
@time chains_NUTS = sample(model, NUTS(), 1000)
```

100%

Another cell defining chains_NUTS contains errors.

```
plot(chains_NUTS; colordim=:parameter, legend=true)
```

1.5 Sampling via PG+NUTS.

```
@time chains_PG_NUTS =
```

	iteration	chain	μ[1]	μ[2]	w[1]	w[2]
1	1	1	-1.94838	-0.773854	0.506241	0.493759
2	2	1	-1.96032	-0.760893	0.502367	0.497633
3	3	1	-1.97654	-0.753359	0.499932	0.500068
4	4	1	-2.14837	-0.676939	0.481817	0.518183
5	5	1	-2.17949	-0.554959	0.49149	0.50851
6	6	1	-2.21794	-0.53658	0.48612	0.51388
7	7	1	-2.26144	-0.570986	0.483773	0.516227
8	8	1	-2.3283	-0.478795	0.50734	0.49266
9	9	1	-2.73912	-0.0723776	0.581018	0.418982
10	10	1	-2.79475	-0.00475192	0.57743	0.42257

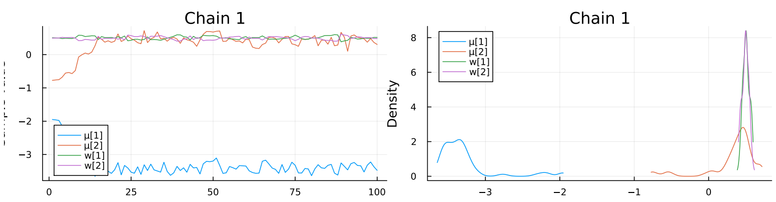
more

```
@time chains_PG_NUTS = sample(model, Gibbs(PG(100, :k),
NUTS(200, 0.65, init_ϵ=0.003, :μ, :w)), 100)

```

```
100%
```

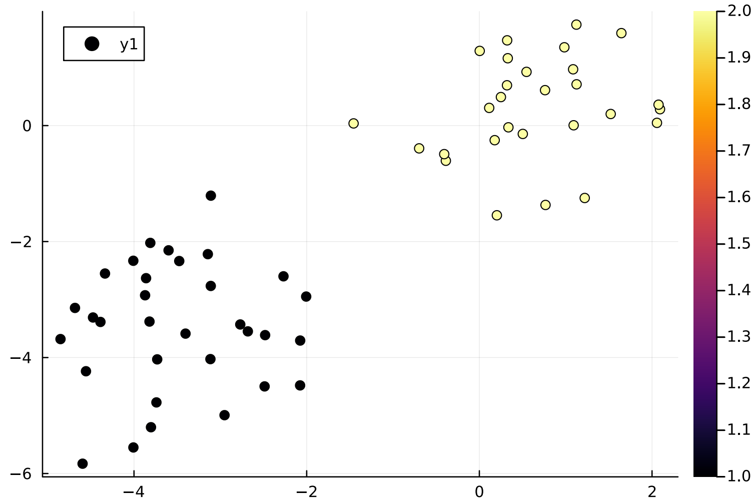
```
194.602903 seconds (426.31 M allocations: 35.779 GiB, 4.84% gc time, 2.60% compilation time)
```




```
plot(chains_PG_NUTS[["μ[1]", "μ[2]", "w[1]", "w[2]"]];
colordim=:parameter, legend=true)
```

```
3-dimensional AxisArray{Float64,3,...} with axes:
    :iter, 1:1:100
    :var, [Symbol("μ[1]"), Symbol("μ[2]"), Symbol("w[1]"), Symbol
    :chain, 1:1
And data, a 100×65×1 Array{Float64, 3}:
[:, :, 1] =
 -1.94838  -0.773854  0.506241  0.493759  …  1.0  1.0  1.0  2.0
 -1.96032  -0.760893  0.502367  0.497633     1.0  1.0  1.0  2.0
 -1.97654  -0.753359  0.499932  0.500068     1.0  1.0  1.0  2.0
 -2.14837  -0.676939  0.481817  0.518183     1.0  1.0  1.0  2.0
 -2.17949  -0.554959  0.49149   0.50851      1.0  1.0  1.0  2.0
 -2.21794  -0.53658   0.48612   0.51388   …  1.0  1.0  1.0  2.0
 -2.26144  -0.570986  0.483773  0.516227     1.0  1.0  1.0  2.0
  ⋮                                       ⋱                 ⋮
 -3.33896   0.495269  0.471419  0.528581     1.0  1.0  1.0  2.0
 -3.5244    0.474689  0.495744  0.504256  …  1.0  1.0  1.0  2.0
 -3.32019   0.373059  0.488997  0.511003     1.0  1.0  1.0  2.0
 -3.22497   0.483376  0.489564  0.510436     1.0  1.0  1.0  2.0
 -3.3667    0.36694   0.517047  0.482953     1.0  1.0  1.0  2.0
 -3.47571   0.304172  0.515882  0.484118     1.0  1.0  1.0  2.0
```

```
chains_PG_NUTS.value
```




```
begin
    scatter(x[1, :], x[2, :];
    marker_z=chains_PG_NUTS.value[iter=100][5:64,1])
    # markerfacecolor=map(color_func, color_vec),
    title="Synthetic Dataset")
end
```