Turing quickstart Beta-Bernoulli

```
#    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)
}
"""
```

```
begin
    using Markdown
    using InteractiveUtils
    using Pkg, DrWatson, PlutoUI
end
```

▸ TaskLocalRNG()

```
begin
    using Distributions
    using FillArrays
    using StatsPlots

    using LinearAlgebra
    using Random
    using Turing

    # Import all libraries.
    using DataFrames
    #DirichletProcess, ChineseRestaurantProcess,
    StickBreakingProcess
    using Turing.RandomMeasures
    # DocStringExtensions provides $(SIGNATURES)
    using DocStringExtensions
    # Set a random seed.
    Random.seed!(3)
end
```

```
using DynamicPPL, Printf
```

Table of Contents


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

```julia
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 virtual cores)
Environment:
  JULIA_PKG_SERVER = https://mirrors.tuna.tsinghua.edu.cn/julia
  JULIA_REVISE_WORKER_ONLY = 1
```

```
"foo bar   34.567"
```

```julia
begin
    x = "foo"
    println("$x $(x) ",  "haha")
    #using Printf
    @sprintf "%s %s %8.3f" "foo" "bar" 34.567
end
```

```
foo foo haha
```

1. Turing.jl

- https://turinglang.org/
- https://turinglang.org/DynamicPPL.jl/stable/
- https://github.com/TuringLang/Turing.jl

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

```
@model function gdemo(n)
    μ ~ Normal(0, 1)
    x ~ MvNormal(Fill(μ, n), I)
end
```

```
▸ [0.848878, -0.319361, -1.39821, -0.0519893, -1.14651, -0.630617,
```

```
begin
    Random.seed!(1776)
    dataset1_1 = randn(100)
end
```

```
▸ [0.71737]
```

```
Random.seed!(16); dataset1_2 = randn(1)
```

1.1 Conditioning and Deconditioning

Bayesian models can be transformed with two main operations, conditioning and deconditioning (also known as marginalization).

Conditioning takes a variable and fixes its value as known. We do this by passing a model and a collection of conditioned variables to | or its alias condition.

1.1.1 `model1_1`: gdemo conditioned on `dataset1_1` and μ=0

- This conditional distribution contains no random variables to draw. Drawing from it returns empty.
- This model contains both data and parameter values.

```
md"### 1.1.1 `model1_1`: gdemo conditioned on `dataset1_1`
and μ=0

- This conditional distribution contains no random
variables to draw. Drawing from it returns empty.
- This model contains both data and parameter values."
```

```
m1_1 =
▸ Model(gdemo (generic function with 2 methods), (n = 100), (), Co
```

```
m1_1 = gdemo(length(dataset1_1)) | (x=dataset1_1, μ=0)
```

This operation can be reversed by applying decondition

```
Model(gdemo (generic function with 2 methods), (n = 100), (), De
```

```
DynamicPPL.decondition(m1_1)
```

- Drawing from it returns empty

```
()
```

```
rand(m1_1)
```

We often want to calculate the (unnormalized) probability density for an event. This probability might be a prior, a likelihood, or a posterior (joint) density. DynamicPPL provides convenient functions for this. For example, we can calculate the joint probability of a set of samples (here drawn from the prior) with logjoint:

```
-151.5171285609121
```

```
begin
    Random.seed!(124)
    sample1_1 = rand(m1_1)
    logjoint(m1_1, sample1_1)
end
```

```
()
```

```
sample1_1
```

For models with many variables `rand(model)` can be prohibitively slow since it returns a `NamedTuple` of samples from the prior distribution of the unconditioned variables. We recommend working with samples of type DataStructures.OrderedDict in this case:

```
-151.5171285609121
```

```
begin
    using DataStructures

    Random.seed!(124)
    @show sample_dict = rand(OrderedDict, m1_1)
    logjoint(m1_1, sample_dict)
end
```

```
sample_dict = rand(OrderedDict, m1_1) = OrderedCollections.OrderedDict{Any, Any}()
```

```
OrderedDict()
```

```
sample_dict
```

- If sample11/*sample*dict is empty, what are logjoint and etc. calculating?
- It is computing the joint of the data & parameters already embedded in the model.

```
md"- If sample1_1/sample_dict is empty, what are logjoint
and etc. calculating?
- It is computing the joint of the data & parameters
already embedded in the model."
```

true

```
begin
    @show logjoint(m1_1, sample1_1)
    @show loglikelihood(m1_1, sample1_1)
    @show logprior(m1_1, sample1_1)
    logjoint(m1_1, sample1_1) ≈ loglikelihood(m1_1,
    sample1_1) + logprior(m1_1, sample1_1)
end
```

```
logjoint(m1_1, sample1_1) = -151.5171285609121
loglikelihood(m1_1, sample1_1) = -151.5171285609121
logprior(m1_1, sample1_1) = 0.0
```

-151.5171285609121

```
@time logjoint(m1_1, sample_dict)
```

```
  0.000006 seconds (4 allocations: 1.109 KiB)
```

m1_1_1 =

▶ Model(gdemo (generic function with 2 methods), (n = 1), (), Conc

```
m1_1_1 = gdemo(length(dataset1_2)) | (x=dataset1_2, μ=15)
```

0.0

```
begin
    @show s1_1_1 = rand(m1_1_1)
    @show logjoint(m1_1_1, s1_1_1)
    @show loglikelihood(m1_1_1, s1_1_1)
    @show logprior(m1_1_1, s1_1_1)
end
```

```
s1_1_1 = rand(m1_1_1) = NamedTuple()
logjoint(m1_1_1, s1_1_1) = -216.33463747881498
loglikelihood(m1_1_1, s1_1_1) = -216.33463747881498
logprior(m1_1_1, s1_1_1) = 0.0
```

Sometimes it is helpful to define convenience functions for conditioning on some variable(s). For instance, in this example we might want to define a version of gdemo that conditions on one observation of x:

gdemo_x (generic function with 1 method)

```
# this function is essentially same as gdemo2.
gdemo_x(x::AbstractVector{<:Real}) = gdemo(length(x)) | (;
x)
```

1.1.2 Correct syntax to define a conditional distribution with a NamedTuple

▶ (μ = -0.668001)

```
Random.seed!(124); rand(gdemo(length(dataset1_1)) |
(x=dataset1_1,) )
```

▶ (μ = -0.668001)

```
Random.seed!(124); rand(gdemo(length(dataset1_1)) |
(;x=dataset1_1,) )
```

▶ (μ = -0.668001)

```
Random.seed!(124); rand(gdemo(length(dataset1_1)) |
(;x=dataset1_1) )
```

Following syntax is wrong in defining a conditional distribution.

- `(x=dataset1_1)` is not a NamedTuple. It defines a variable named x.

```
Random.seed!(124) ; rand(gdemo(length(dataset1_1)) | (x=dat
aset1_1) )
```

1.1.3 Sample from a fake conditional distribution (a full distribution in fact)

The following is NOT a conditional distribution as dataset1_1 is not a symbol in the model. It effectively is the full distribution. Check the sampling result in the next block.

```
m1_3 =
▸ Model(gdemo (generic function with 2 methods), (n = 100), (), Co
```

```
1 m1_3 = gdemo(length(dataset1_1)) | (; dataset1_1)
```

```
▸ (μ = -0.389096, x = [-0.805381, -2.01082, -0.583562, -0.788156,
```

```
1 rand(m1_3)
```

1.1.4 Sample μ from gdemo() conditioned on an x.

- This conditional distribution of μ is NOT μ posterior, but the prior distribution.
- Evidenced from the μ histogram.

```
▸ [0.71737]
```

```
1 dataset1_2
```

```
m1_4 =
▸ Model(gdemo (generic function with 2 methods), (n = 1), (), Conc
```

```
1 m1_4 = gdemo(length(dataset1_2)) | (;x=dataset1_2 )
```

```
▸ OrderedDict(μ ⇒ 0.0419453)
```

```
1 @time rand(OrderedDict, m1_4)
```

```
  0.021545 seconds (35.39 k allocations: 2.442 MiB,
99.55% compilation time)
```

```
▸ (μ = -2.28973)
```

```
1 @time rand(m1_4)
```

```
  0.000061 seconds (36 allocations: 2.547 KiB)
```

```
-0.22876723439327667
```

```
1 rand(OrderedDict, m1_4).vals[1]
```

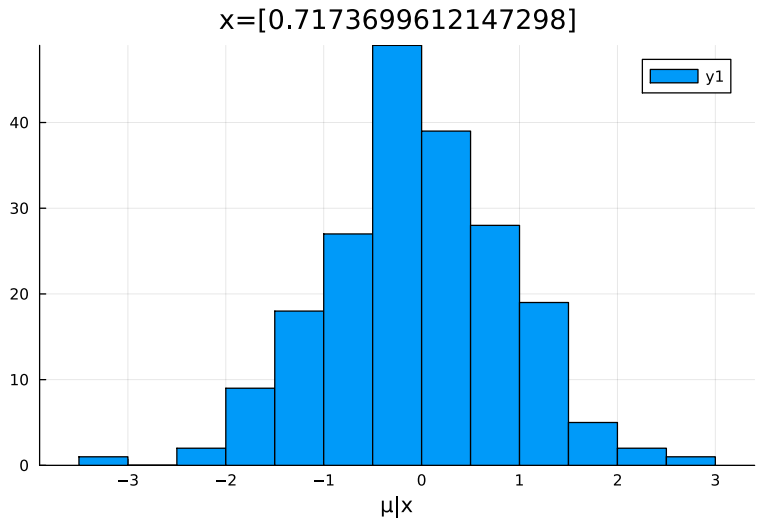
```
n_samples = 200
```

```
1 n_samples = 200
```

▸ [0.540481, -1.0629, -0.729338, 0.259889, 0.261761, 0.555702, 0.0

```
@time begin
    # draw μ repeatedly and draw a histogram
    μ_posterior_vec = Vector{Float64}(undef, n_samples)
    for i ∈ 1:n_samples
        sample_dict = rand(OrderedDict, m1_4)
        μ_posterior_vec[i] = sample_dict.vals[1]
    end
    μ_posterior_vec
end
```

```
  0.004735 seconds (6.22 k allocations: 455.969 Ki
B, 81.21% compilation time)
```




```
histogram(μ_posterior_vec, xlabel="μ|x",
title="x=$(dataset1_2)")
```

1.1.5 Sample x from gdemo() conditioned on an μ=15.

- This truly is a conditional distribution, evidenced by the histogram of sampled x

```
m1_5 =
▸ Model(gdemo (generic function with 2 methods), (n = 1), (), Conc
```

```
m1_5 = gdemo(1) | (;μ=15)
```

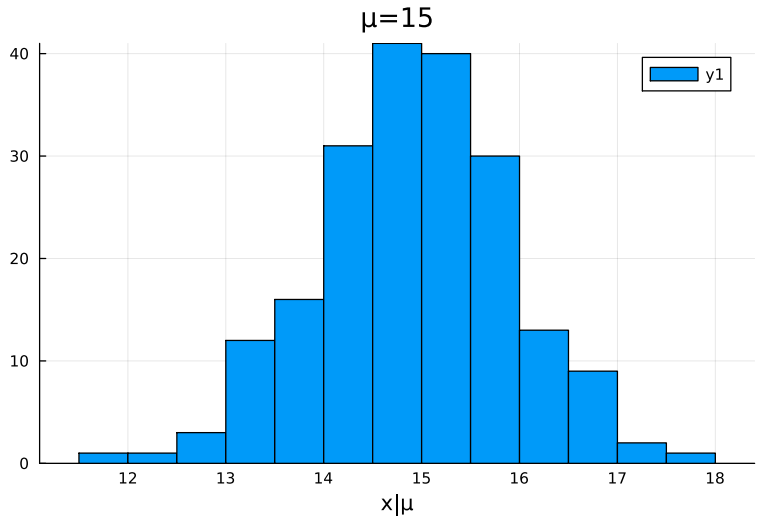
16.318092777878245

```
rand(OrderedDict, m1_5).vals[1][1]
```

▸ [14.8499, 13.4372, 14.3674, 13.5304, 15.7144, 14.9822, 15.2636,

```
@time begin
    # draw repeatedly and draw a histogram
    x_posterior_vec = Vector{Float64}(undef, n_samples)
    for i ∈ 1:n_samples
        sample_dict = rand(OrderedDict, m1_5)
        x_posterior_vec[i] = sample_dict.vals[1][1]
    end
    x_posterior_vec
end
```

```
  0.000913 seconds (6.20 k allocations: 473.641 Ki
B)
```




```
histogram(x_posterior_vec, xlabel="x|μ", title="μ=15")
```

- likelihood and prior are **reversed** due to x not being the argument of gdemo()

true

```
begin
    @show logjoint(m1_5, (;x=dataset1_2))
    @show loglikelihood(m1_5, (;x=dataset1_2))
    @show logprior(m1_5, (x=dataset1_2,))
    logjoint(m1_5, (;x=dataset1_2)) ≈ loglikelihood(m1_5,
    (;x=dataset1_2)) + logprior(m1_5, (x=dataset1_2,))
end
```

```
logjoint(m1_5, (; x = dataset1_2)) = -216.334637478
81498
loglikelihood(m1_5, (; x = dataset1_2)) = -113.4189385
3320467
logprior(m1_5, (x = dataset1_2,)) = -102.9156989456103
2
```

- likelihood of x given μ=15

-102.92098303320466

```
logpdf(Normal(15,1), 0.717)
```

- prior of μ=15 given μ ~ N(0,1)

-113.41893853320467

```
logpdf(Normal(0,1), 15)
```

-216.33463747881498

```
let
    @show logjoint(gdemo(1), (;x=dataset1_2, μ=15))
    @show loglikelihood(gdemo(1), (;x=dataset1_2, μ=15))
    @show logprior(gdemo(1), (x=dataset1_2, μ=15))
end
```

```
logjoint(gdemo(1), (; x = dataset1_2, μ = 15)) = -216.33463747881498
loglikelihood(gdemo(1), (; x = dataset1_2, μ = 15)) = 0.0
logprior(gdemo(1), (x = dataset1_2, μ = 15)) = -216.33463747881498
```

- Likelihood=0!
- Prior = Joint = Likelihood of x + prior of μ.

1.1.6 gdemo2(): x(observations) as input argument

gdemo2 (generic function with 2 methods)

```
@model function gdemo2(x)
    μ ~ Normal(0, 1)
    x ~ MvNormal(Fill(μ, length(x)), I)
end
```

-113.41893853320467

```
let
    @show logjoint(gdemo2([0.717]), (;x=dataset1_2, μ=15))
    @show loglikelihood(gdemo2([0.717]), (;x=dataset1_2, μ=15))
    @show logprior(gdemo2([0.717]), (x=dataset1_2, μ=15))
end
```

```
logjoint(gdemo2([0.717]), (; x = dataset1_2, μ = 15)) = -216.33992156640932
loglikelihood(gdemo2([0.717]), (; x = dataset1_2, μ = 15)) = -102.92098303320466
logprior(gdemo2([0.717]), (x = dataset1_2, μ = 15)) = -113.41893853320467
```

- Likelihood and prior are now correct.

-113.41893853320467

```
let
    @show logjoint(gdemo2([0.717]), (;μ=15))
    @show loglikelihood(gdemo2([0.717]), (; μ=15))
    @show logprior(gdemo2([0.717]), ( μ=15,))
end
```

```
logjoint(gdemo2([0.717]), (; μ = 15)) = -216.33992156640932
loglikelihood(gdemo2([0.717]), (; μ = 15)) = -102.92098303320466
logprior(gdemo2([0.717]), (μ = 15,)) = -113.41893853320467
```

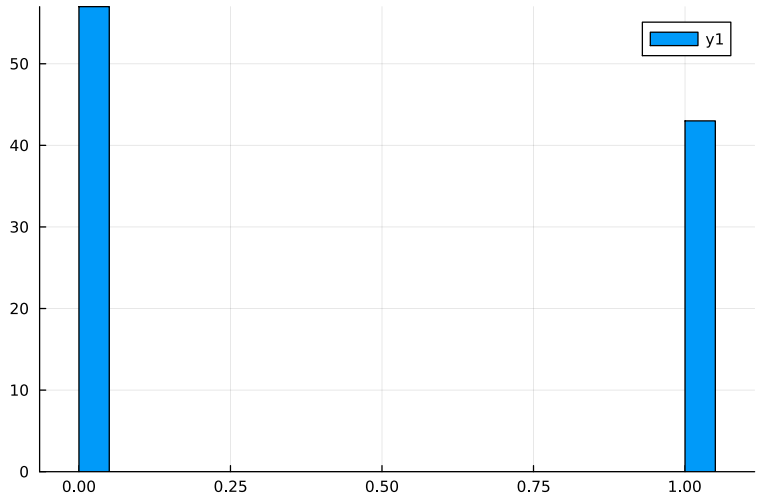
2. The Problem: Coin flipping

Flip a coin. A random variable x = 1 if the coin is head up. x=0 if the coin is tail up.

2.1 Question

- Input: a sequence of observed x.
- Output: what is the p(head up) of the coin?

2.2 Input: flip 100 times




```
begin
    # Set the true probability of heads in a coin.
    p_true = 0.5

    # Iterate from having seen 0 observations to 100
    observations.
    Ns = 0:100

    # Fix the seed and draw data from a Bernoulli
    distribution, i.e. draw heads or tails.
    Random.seed!(12)
    data = rand(Bernoulli(p_true), last(Ns))

    println("The number of ups is
    $(sum(data))/$(length(data))." );
    histogram(data, bins=20)
end
```

```
The number of ups is 43/100.
```

3 Frequentist/Likelihood approach

p is an unknown parameter to be solved.

$x_i \sim$ Bernoulli(p)

The MLE (maximum likelihood estimator) is $\hat{p} = \sum_{i=1}^{i=N} x_i/N$ and why?

- What is the difference between likelihood and probability? What is MLE?
- Why most estimators we know/use are MLE estimators?
- Analytic/closed-form （解析） solution vs Monte Carlo solution vs Brute force solution?
    - https://en.wikipedia.org/wiki/Closed-form_expression

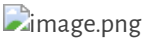
```
md"# 3 Frequentist/Likelihood approach

p is an unknown parameter to be solved.

xᵢ ~ Bernoulli(p)

The MLE (maximum likelihood estimator) is p̂ = ∑ᵢ₌₁^{i=N} xᵢ/N
and why?

- What is the difference between likelihood and
probability? What is MLE?
- Why most estimators we know/use are MLE estimators?
- Analytic/closed-form (解析) solution vs Monte Carlo
solution vs Brute force solution?
  - https://en.wikipedia.org/wiki/Closed-form_expression"
```

For any model (Likelihood or Bayesian):

- Is there an analytic solution? No for most real-world problems.
- Is there a monte carlo solution? Yes for most real-world problems.
- Is there a brute-force solution? Yes. But may take hundreds of years.

3.1 The MLE (maximum likelihood estimator) is $\hat{p} = \sum_{i=1}^{i=N} x_i/N$ and why?

- https://stats.stackexchange.com/questions/275380/maximum-likelihood-estimation-for-bernoulli-distribution



```
md"## 3.1 The MLE (maximum likelihood estimator) is  p̂ = ∑ᵢ₌₁^{i=N} xᵢ/N and why?


- https://stats.stackexchange.com/questions/275380/maximum-likelihood-estimation-for-bernoulli-distribution

![image.png](attachment:f0a44559-d8a5-4842-8ffa-fb2a6d31481f.png)"
```

4 Bayesian

p is a random variable with a prior distribution.

$$p \sim Beta(\alpha, \beta)$$

$$x_i \sim Bernoulli(p)$$

Note: x is Bernoulli, 0 or 1, not Binomial.

Question: derive the posterior distribution of p, AKA (Also Known As), $Prob(p|x_1, \ldots x_N)$.

```
md"# 4 Bayesian

p is a random variable with a prior distribution.

$p \sim Beta(\alpha, \beta)$

$x_i \sim Bernoulli(p)$

Note: x is Bernoulli, 0 or 1, not Binomial.

Question: derive the posterior distribution of $p$, AKA (Also Known As), $Prob(p|x_1, ... x_N)$."
```

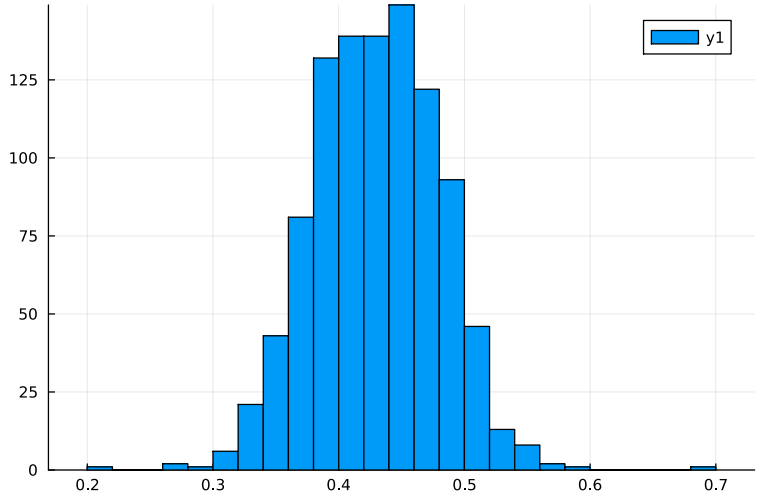
- Is there an analytic solution to this? Yes, Beta-Binomial.
  - https://en.wikipedia.org/wiki/Beta-binomial_distribution

- Is there a monte carlo solution? Yes, we will talk about this today.

4.1 Declare our Bayesian model and Monte Carlo with a HMC sampler

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

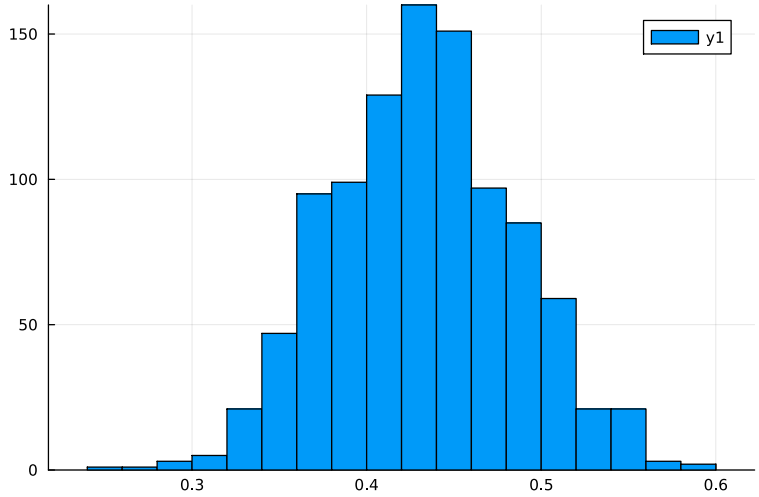
```
# Declare our Bayesian model,p ~ beta(1,1), x ~ Bernoilli(p)
@model function coinflip(x)
    # Our prior belief about the probability of heads in a coin.
    p ~ Beta(1, 1)

    # The number of observations.
    N = length(x)
    for n in 1:N
        # Heads or tails of a coin are drawn from a Bernoulli distribution.
        x[n] ~ Bernoulli(p)
    end
end
```




```
begin
    # Settings of the Hamiltonian Monte Carlo (HMC) sampler.
    iterations = 1000
    ϵ = 0.05
    τ = 10

    # Start sampling.
    @time chain = sample(coinflip(data), HMC(ϵ, τ),
    iterations)
    display(chain)
    # Plot a summary of the sampling process for the
    parameter p, i.e. the probability of heads in a coin.
    histogram(chain[:p])
end
```

100%

```
 13.902008 seconds (9.63 M allocations: 653.702 Mi
B, 2.78% gc time, 64.55% compilation time)
Chains MCMC chain (1000×11×1 Array{Float64, 3}):

Iterations        = 1:1:1000
Number of chains  = 1
Samples per chain = 1000
Wall duration     = 10.75 seconds
Compute duration  = 10.75 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse    ess_bulk
ess_tail      rh …
      Symbol   Float64   Float64   Float64     Float64
Float64   Float …

           p    0.4303    0.0499    0.0009   3000.0000
421.4425    0.99 …

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.3377    0.3950    0.4304    0.4668
0.5197
```




```
begin
    # Using NUTS as sampler
    @time chain_NUTS = sample(coinflip(data), NUTS(),
    iterations)
    display(chain_NUTS)
    # Plot a summary of the sampling process for the
    parameter p, i.e. the probability of heads in a coin.
    histogram(chain_NUTS[:p])
end
```

100%

Found initial step size
ϵ: 0.8

```
  6.243953 seconds (4.15 M allocations: 283.049 Mi
B, 1.66% gc time, 56.34% compilation time)
Chains MCMC chain (1000×13×1 Array{Float64, 3}):

Iterations        = 501:1:1500
Number of chains  = 1
Samples per chain = 1000
Wall duration     = 5.16 seconds
Compute duration  = 5.16 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, max_hamiltonian_energy_error, tree_depth,
numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse   ess_bulk
ess_tail      rha ⋯
      Symbol   Float64   Float64   Float64    Float64
Float64   Float6 ⋯

           p    0.4330    0.0524    0.0022   578.1231
695.4015    0.999 ⋯

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.3355    0.3954    0.4333    0.4673
0.5412
```

4.1.1 The posterior of p is max at $ p=0.4311$, almost same as MLE. Not 0.5!

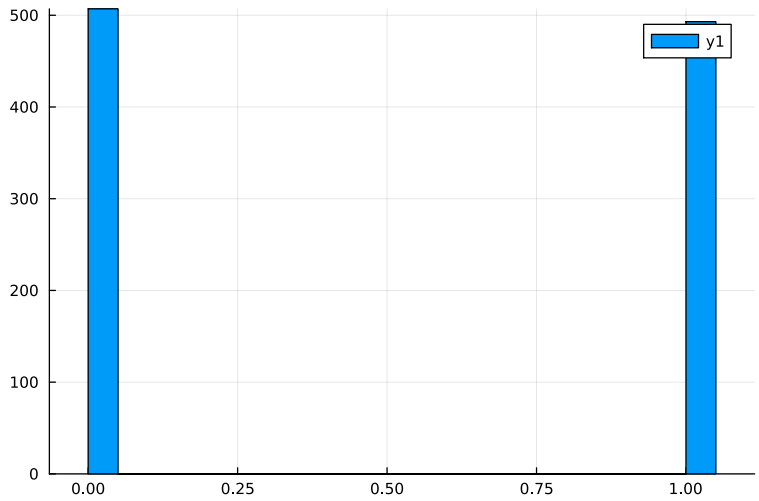
Because:

1. the data is 43% up;
2. we used a flat prior (uniform $\in [0, 1]$).

```
md"### 4.1.1 The posterior of p is max at $ p=0.4311$,
almost same as MLE. Not 0.5!
Because:
1) the data is 43% up;
2) we used a flat prior (uniform $\in [0,1]$). "
```

4.2 Input is 10X more data.

```
md"## 4.2 Input is 10X more data."
```

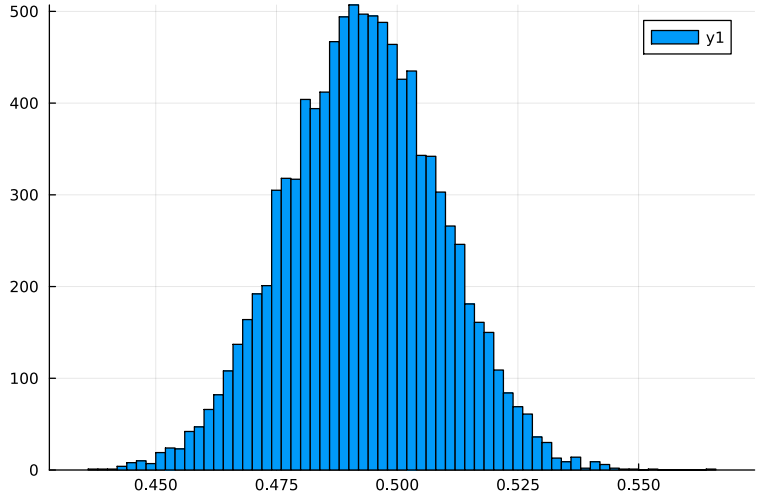



```
begin
    # Fix the seed and draw data from a Bernoulli
    distribution, i.e. draw heads or tails.
    Random.seed!(7)
    data2 = rand(Bernoulli(p_true), 1000)
    println("The number of ups is
    $(sum(data2))/$(length(data2)) ");
    histogram(data2, bins=20)
end
```

```
The number of ups is 493/1000
```

4.2.1 sample using the same model but with 10X iterations

```
md"### 4.2.1 sample using the same model but with 10X
iterations"
```



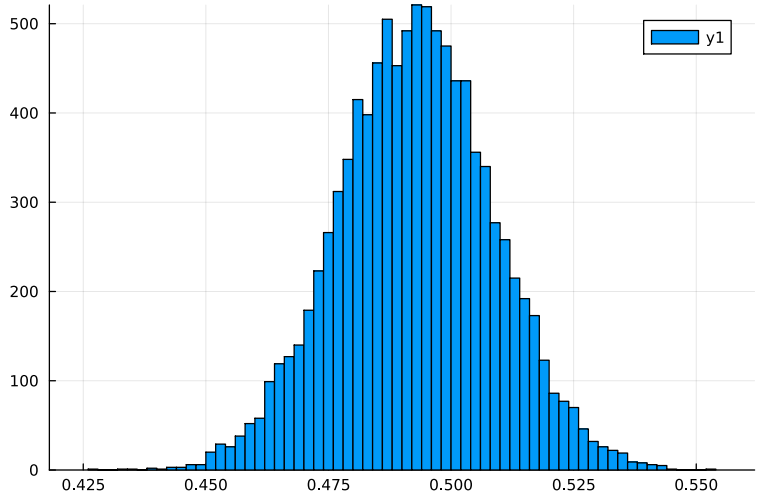

```
begin
    # sampling using the same model but more iterations
    #ϵ=0.05, τ=10
    @time chain_10X_HMC = sample(coinflip(data2), HMC(0.05,
    10), 10_000)
    display(chain_10X_HMC)
    # Plot a summary of the sampling process for the
    parameter p, i.e. the probability of heads in a coin.
    histogram(chain_10X_HMC[:p])
end
```

100%

```
  3.998461 seconds (5.17 M allocations: 274.212 Mi
B, 2.04% gc time)
Chains MCMC chain (10000×11×1 Array{Float64, 3}):

Iterations        = 1:1:10000
Number of chains  = 1
Samples per chain = 10000
Wall duration     = 3.93 seconds
Compute duration  = 3.93 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse     ess_bul
k     ess_tail      ⋯
      Symbol   Float64   Float64   Float64      Float6
4     Float64   Fl ⋯

           p    0.4930    0.0159    0.0001   15496.752
1   10138.5250    1 ⋯

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.4619    0.4821    0.4930    0.5037
0.5240
```

4.2.2 same model, but using the NUTS sampler (default parameters)

```
md"### 4.2.2 same model, but using the NUTS sampler
(default parameters)"
```




```
begin
    # Start sampling.
    @time chain_10X_NUTS = sample(coinflip(data2), NUTS(),
    10_000)
    display(chain_10X_NUTS)
    # Plot a summary of the sampling process for the
    parameter p, i.e. the probability of heads in a coin.
    histogram(chain_10X_NUTS[:p])
end
```

100%

Found initial step size
ϵ: 0.2

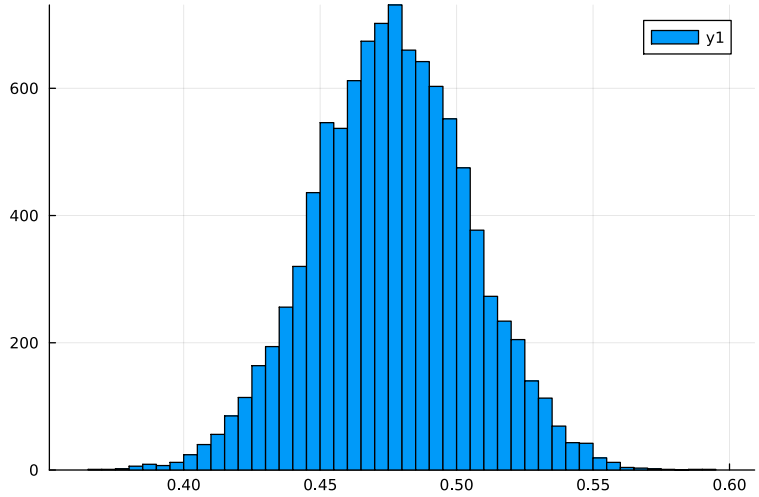
```
  6.295829 seconds (4.19 M allocations: 276.446 Mi
B, 1.45% gc time)
Chains MCMC chain (10000×13×1 Array{Float64, 3}):

Iterations        = 1001:1:11000
Number of chains  = 1
Samples per chain = 10000
Wall duration     = 6.18 seconds
Compute duration  = 6.18 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, max_hamiltonian_energy_error, tree_depth,
numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse    ess_bulk
ess_tail      r …
      Symbol   Float64   Float64   Float64     Float64
Float64   Floa …

           p    0.4928    0.0157    0.0002   4407.9601
6543.2979    1.0 …

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.4622    0.4821    0.4928    0.5033
0.5238
```

4.3 Modify the prior to alter the posterior

```
md"## 4.3 Modify the prior to alter the posterior"
```

```
coinflip_prior_around_0_5 (generic function with 2 methods)
```

```
# Declare our Bayesian model,p ~ beta, x ~ Bernoilli(p)
# But let's modify the \alpha and \beta of the prior
distribution.

@model function coinflip_prior_around_0_5(x)
    # Our prior belief about the probability of heads in a
coin.
    p ~ Beta(100, 100)

    # The number of observations.
    N = length(x)
    for n in 1:N
        # Heads or tails of a coin are drawn from a
Bernoulli distribution.
        x[n] ~ Bernoulli(p)
    end
end
```




```
begin
    # Start sampling.
    @time chain_strong_beta_NUTS =
    sample(coinflip_prior_around_0_5(data), NUTS(), 10_000)
    display(chain_strong_beta_NUTS)
    # Plot a summary of the sampling process for the
    parameter p, i.e. the probability of heads in a coin.
    histogram(chain_strong_beta_NUTS[:p])
end
```

100%

Found initial step size
ϵ: 0.4

```
  4.004209 seconds (5.56 M allocations: 373.837 Mi
B, 3.37% gc time, 53.42% compilation time)
Chains MCMC chain (10000×13×1 Array{Float64, 3}):

Iterations        = 1001:1:11000
Number of chains  = 1
Samples per chain = 10000
Wall duration     = 3.03 seconds
Compute duration  = 3.03 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, max_hamiltonian_energy_error, tree_depth,
numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse    ess_bulk
ess_tail      r …
      Symbol   Float64   Float64   Float64     Float64
Float64   Floa …

           p    0.4765    0.0284    0.0004   4071.7624
6847.0813    1.0 …

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.4203    0.4571    0.4764    0.4955
0.5331
```

4.3.1 Conclusion

- With a Strong prior beta(100,100) around 0.5, we can obtain a posterior estimate around 0.48, instead of 0.43.

```
md"### 4.3.1 Conclusion
- With a Strong prior beta(100,100) around 0.5, we can
obtain a posterior estimate around 0.48, instead of 0.43."
```

5 Test two different Negative Binomial models

1. with a beta prior for p and Uniform prior for r
2. with a beta prior for p and Gamma prior for r

```
md"# 5 Test two different Negative Binomial models
1. with a beta prior for p and Uniform prior for r
1. with a beta prior for p and Gamma prior for r "
```

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

```
begin
    # Generate synthetic data with known parameters
    Random.seed!(28)
    n = 1000
    r = 3
    p = 0.2
    y = rand(NegativeBinomial(r, p), n)

    # Bayesian Negative binomial model
    @model nb_bayes(y) = begin
        # Defining variables
        r ~ Uniform(0, 100)
        p ~ Beta(2, 2)

        # Likelihood function
        y_dist = NegativeBinomial(r, p)
        y ~ y_dist
    end


    # Define the negative binomial likelihood
    @model function negative_binomial_model(y)
        # Prior distribution for the success probability
        p ~ Beta(1, 1)
        # Prior distribution for the dispersion parameter
        r ~ Gamma(1, 1)
        y_dist = NegativeBinomial(r, p)
        # Likelihood
        for i in eachindex(y)
            y[i] ~ y_dist
        end
    end

end
```

▸ (1000)

```
size(y)
```

5.1 NB with a beta prior for p and Uniform prior for r

```
md"## 5.1 NB with a beta prior for p and Uniform prior for
r"
```

```
begin
    # Sampling from the posterior distribution
    @time chain_w_uniform_and_beta_prior =
    sample(nb_bayes(y), NUTS(), 1_000)
    display(chain_w_uniform_and_beta_prior)
end
```

```
100%
```

```
Found initial step size
ϵ: 0.0125
```

```
 11.588600 seconds (8.27 M allocations: 571.869 M
iB, 2.20% gc time, 63.28% compilation time)
Chains MCMC chain (1000×14×1 Array{Float64, 3}):

Iterations        = 501:1:1500
Number of chains  = 1
Samples per chain = 1000
Wall duration     = 9.76 seconds
Compute duration  = 9.76 seconds
parameters        = r, p
internals         = lp, n_steps, is_accept, acceptan
ce_rate, log_density, hamiltonian_energy, hamiltonia
n_energy_error, max_hamiltonian_energy_error, tree_d
epth, numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse   ess_bul
k   ess_tail      rha …
      Symbol   Float64   Float64   Float64    Floate
4    Float64   Floate …

           r    3.0357    0.1669    0.0109   260.173
6   248.9483    1.005 …
           p    0.1988    0.0092    0.0006   253.283
2   276.1358    1.004 …

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           r    2.7451    2.9313    3.0316    3.1235
3.3961
           p    0.1824    0.1929    0.1990    0.2040
```

	parameters	mean	std	mcse	ess_bulk
1	:r	3.03573	0.166911	0.0108527	260.174
2	:p	0.198843	0.00922964	0.000609783	253.283

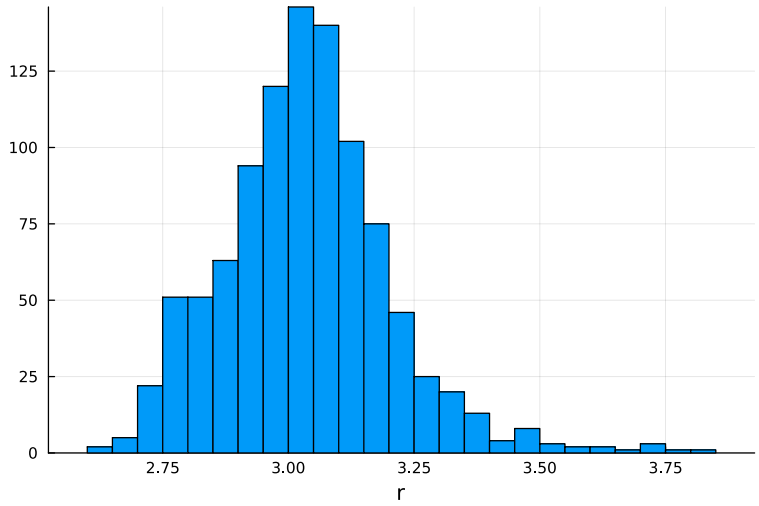
```
# Summarizing posterior distribution
@time describe(chain_w_uniform_and_beta_prior)
```

```
  0.001812 seconds (916 allocations: 909.609 KiB)
```

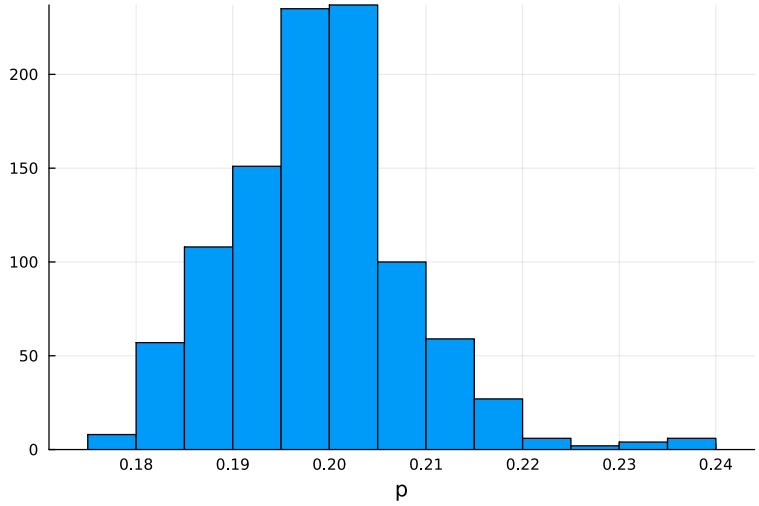
	variable	mean	min	median	ma
1	:iteration	1000.5	501	1000.5	1500
2	:chain	1.0	1	1.0	1
3	:r	3.03573	2.61791	3.03156	3.80
4	:p	0.198843	0.176439	0.198989	0.23
5	:lp	-3363.98	-3371.6	-3363.65	-336
6	:n_steps	7.178	1.0	7.0	31.0
7	:is_accept	1.0	1.0	1.0	1.0
8	:acceptance_rate	0.804302	0.000941705	0.902608	1.0
9	:log_density	-3363.98	-3371.6	-3363.65	-336
10	:hamiltonian_energy	3364.94	3363.03	3364.67	3371.
⋮	more				
16	:nom_step_size	0.346126	0.346126	0.346126	0.34

```
@time describe(DataFrame(chain_w_uniform_and_beta_prior))
```

```
  0.411745 seconds (215.26 k allocations: 15.133 MiB, 99.63% compilation time: 61% of which was recompilation)
```




```
histogram(chain_w_uniform_and_beta_prior[:r], xlabel="r", legend=false)
```




```
histogram(chain_w_uniform_and_beta_prior[:p], xlabel="p",
legend=false)
```

5.2 NB with a beta prior for p and Gamma prior for r

```
md"## 5.2 NB with a beta prior for p and Gamma prior for r"
```

```
@time chain_with_gamma_and_beta_prior =
```

	iteration	chain	p	r	lp	n_steps	is_a
1	501	1	0.208408	3.13897	-3362.46	7.0	1.0
2	502	1	0.199412	3.09801	-3361.88	7.0	1.0
3	503	1	0.188981	2.85574	-3361.71	15.0	1.0
4	504	1	0.201038	3.06947	-3361.43	9.0	1.0
5	505	1	0.200822	3.05335	-3361.43	3.0	1.0
6	506	1	0.192431	2.92301	-3361.44	11.0	1.0
7	507	1	0.202756	3.11294	-3361.57	9.0	1.0
8	508	1	0.185653	2.79	-3362.15	11.0	1.0
9	509	1	0.184202	2.77843	-3362.3	3.0	1.0
10	510	1	0.182481	2.76941	-3362.58	3.0	1.0

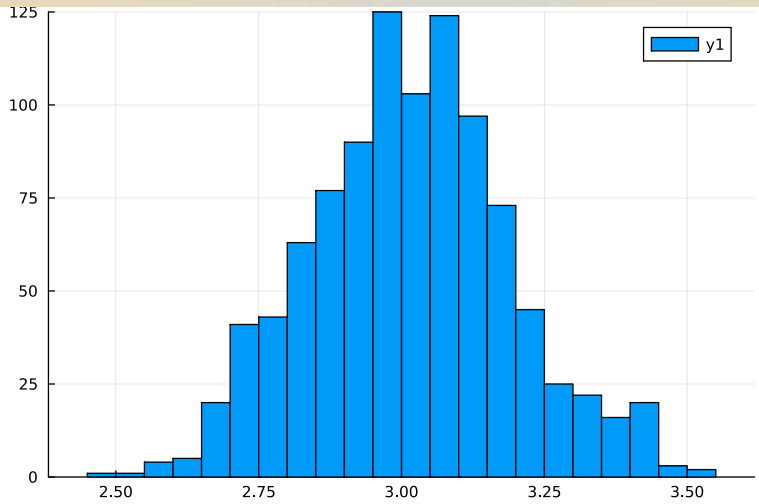
⋮ more

```
@time chain_with_gamma_and_beta_prior =
sample(negative_binomial_model(y), NUTS(0.65), 1000)
```

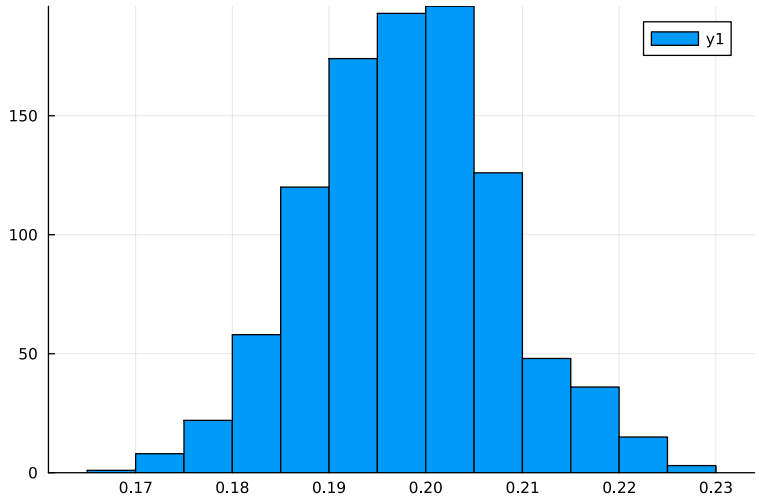
```
100%
```

```
Found initial step size
ϵ: 0.025
```

```
 10.284404 seconds (4.91 M allocations: 343.143 Mi
B, 1.86% gc time, 48.82% compilation time)
```




```
1 histogram(chain_with_gamma_and_beta_prior[:r])
```




```
1 histogram(chain_with_gamma_and_beta_prior[:p])
```

6 Estimate p of Binomial(N, p) from Binomial data with different Ns

```
md"# 6 Estimate p of Binomial(N, p) from Binomial data with different Ns"
```

```
simulate_data (generic function with 4 methods)
```

```
function simulate_data(p::Float64, cov_mean::Int=4000, cov_std::Int=700, no_of_samples::Int=500)
    # Generate synthetic data with known parameters
    # cov_mean, cov_std are the parameters for the normal distribution.

    # fix the seed
    Random.seed!(10)
    # Create a normal distribution centered around 4000
    cov_dist = Normal(cov_mean, cov_std)

    # Sample an integer coverage vector from the distribution
    N_v = round.(rand(cov_dist, no_of_samples))

    x_v = zeros(Int, length(N_v))
    for (i, N) in enumerate(N_v)
        x_v[i] = rand(Binomial(N, p) )
    end
    vaf_v = x_v./N_v
    return N_v, x_v, vaf_v
end
```

```
begin
    @time N_v, x_v, vaf_v = simulate_data(0.0005)
    @time N_v_005, x_v_005, vaf_v_005 = simulate_data(0.005)
    @time N_v_00001, x_v_00001, vaf_v_00001 = simulate_data(0.00001)

    @time println("The number of samples with VAF=0: $(sum(vaf_v_00001.==0) ).");
end
```

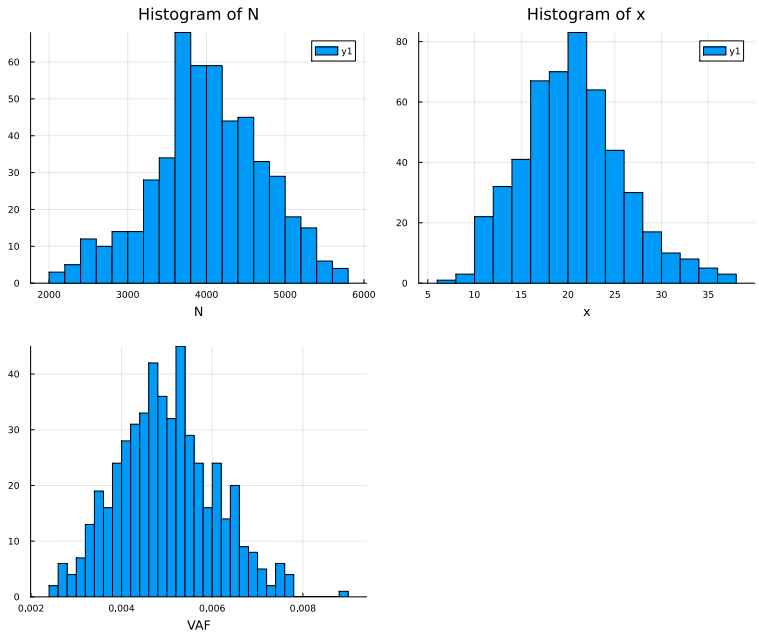
```
  0.000094 seconds (10 allocations: 16.703 KiB)
  0.000097 seconds (10 allocations: 16.703 KiB)
  0.000038 seconds (10 allocations: 16.703 KiB)
The number of samples with VAF=0: 490.
  0.139649 seconds (122.03 k allocations: 8.158 MiB, 18.22% gc time, 99.79% compilation time)
```

```
Vector{Float64} (alias for Array{Float64, 1})
```

```
typeof(vaf_v)
```

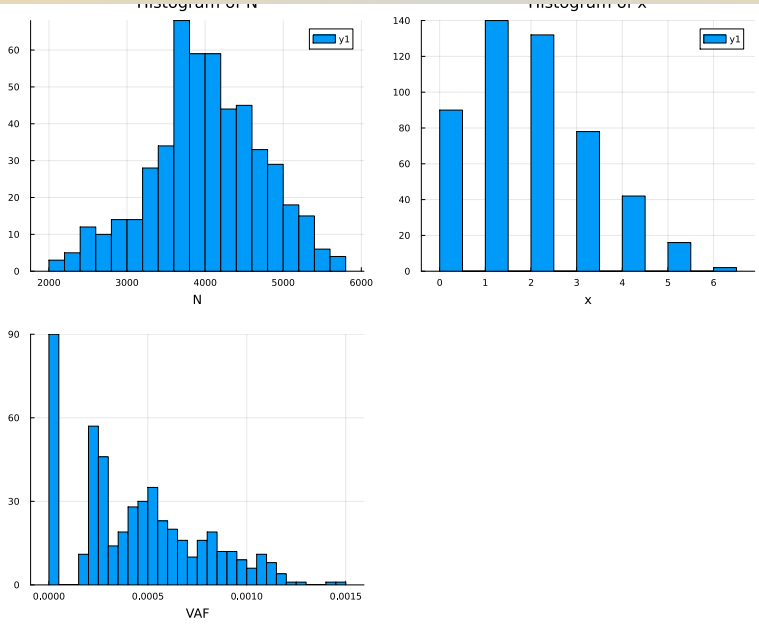
```
plot_histogram (generic function with 1 method)
```

```
function plot_histogram(N_v::Vector{Float64}, x_v::Vector{Int64}, vaf_v::Vector{Float64}, p::Float64)
    @time h1 = histogram(N_v, title="Histogram of N", xaxis="N")
    @time h2 = histogram(x_v, title="Histogram of x", xaxis="x")
    @time h_of_vaf_v = histogram(vaf_v, xaxis="VAF", bins=30, legend=false)
    @time plot(h1, h2, h_of_vaf_v, layout=(2,2), left_margin=5*Plots.mm,
        bottom_margin=5*Plots.mm, size=(1000,830) )
end
```



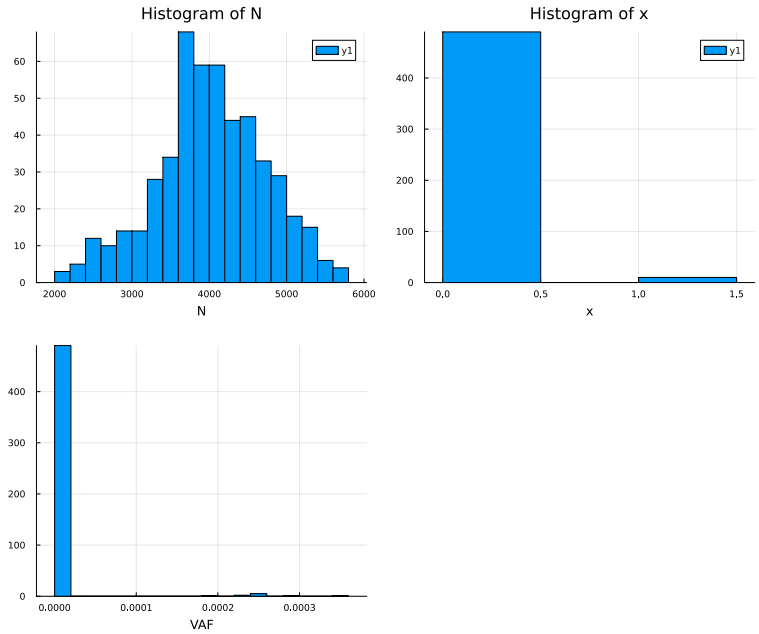

```
plot_histogram(N_v_005, x_v_005, vaf_v_005, 0.005)
```

```
  0.001736 seconds (4.23 k allocations: 234.562 KiB)
  0.001332 seconds (4.20 k allocations: 234.062 KiB)
  0.001255 seconds (4.42 k allocations: 241.750 KiB)
  0.135867 seconds (169.07 k allocations: 11.719 MiB, 98.34% compilation time)
```




```
plot_histogram(N_v, x_v, vaf_v, 0.0005)
```

```
  0.001699 seconds (4.23 k allocations: 234.562 KiB)
  0.001146 seconds (4.15 k allocations: 232.406 KiB)
  0.001224 seconds (4.38 k allocations: 240.594 KiB)
  0.001369 seconds (1.49 k allocations: 301.781 KiB)
```




```
plot_histogram(N_v_00001, x_v_00001, vaf_v_00001, 0.00001)
```

```
  0.001799 seconds (4.23 k allocations: 234.562 KiB)
  0.001162 seconds (4.02 k allocations: 225.906 KiB)
  0.001190 seconds (4.22 k allocations: 225.031 KiB)
  0.001459 seconds (1.49 k allocations: 303.516 KiB)
```

binomial_w_beta_prior (generic function with 2 methods)

```
# Define the model
@model binomial_w_beta_prior(N, x) = begin
    # Prior distribution for the success probability
    p ~ Beta(1, 5)
    # Prior distribution for the dispersion parameter, for
the NB distribution
    # r ~ Gamma(1, 1)
    #y_dist = NegativeBinomial(r, p)
    # Likelihood
    for i in eachindex(N)
        x[i] ~ Binomial(N[i], p)
    end
end
```

6.1 Estimate if true p is 0.0005

```
md"## 6.1 Estimate if true p is 0.0005"
```

```
begin
    # Sampling from the posterior distribution
    @time chain_binomial_w_beta_prior =
    sample(binomial_w_beta_prior(N_v, x_v), NUTS(), 2000)
    display(chain_binomial_w_beta_prior)
end
```

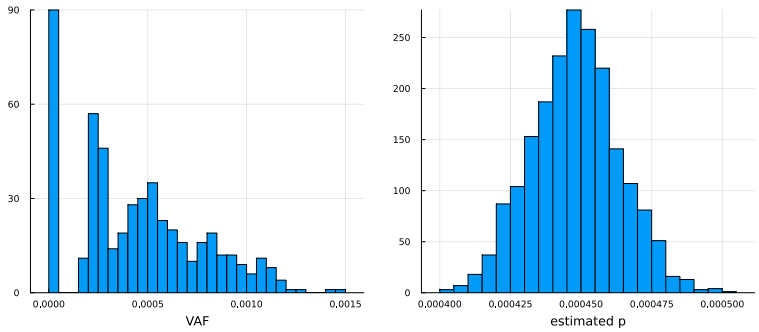
100%

Found initial step size
ϵ: 0.003125

```
  4.232009 seconds (2.62 M allocations: 180.172 Mi
B, 2.46% gc time, 51.77% compilation time)
Chains MCMC chain (2000×13×1 Array{Float64, 3}):

Iterations        = 1001:1:3000
Number of chains  = 1
Samples per chain = 2000
Wall duration     = 3.28 seconds
Compute duration  = 3.28 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, max_hamiltonian_energy_error, tree_depth,
numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse   ess_bulk
ess_tail      rha ⋯
      Symbol   Float64   Float64   Float64    Float64
Float64   Float6 ⋯

           p    0.0004    0.0000    0.0000   882.0261
999.6807    0.999 ⋯

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.0004    0.0004    0.0004    0.0005
0.0005
```

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

```
function plot_estimates(vaf_v, estimated_p)
    h_of_vaf_v = histogram(vaf_v, xaxis="VAF", bins=30, legend=false)
    h_of_p_bin = histogram(estimated_p, xaxis="estimated p", legend=false)
    plot(h_of_vaf_v, h_of_p_bin, layout=(1,2), left_margin=5*Plots.mm, bottom_margin=5*Plots.mm, size=(1000,430) )
end
```




```
plot_estimates(vaf_v, chain_binomial_w_beta_prior[:p])
```

6.2 Estimate if true p is 0.005

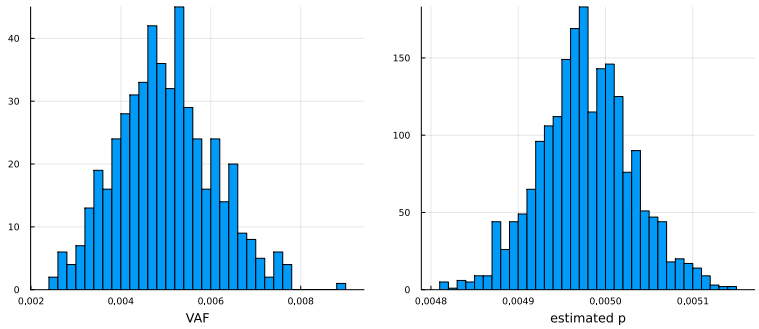
```
md"## 6.2 Estimate if true p is 0.005"
```

```
begin
    # Sampling from the posterior distribution
    @time chain_binomial_w_beta_prior_005 =
    sample(binomial_w_beta_prior(N_v_005, x_v_005), NUTS(),
    2000)
    display(chain_binomial_w_beta_prior_005)
end
```

100%

Found initial step size
ϵ: 0.003125

```
  7.797897 seconds (1.14 M allocations: 76.075 MiB,
0.27% gc time)
Chains MCMC chain (2000×13×1 Array{Float64, 3}):

Iterations        = 1001:1:3000
Number of chains  = 1
Samples per chain = 2000
Wall duration     = 7.78 seconds
Compute duration  = 7.78 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, max_hamiltonian_energy_error, tree_depth,
numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse    ess_bulk
ess_tail      rh …
      Symbol   Float64   Float64   Float64     Float64
Float64   Float …

           p    0.0050    0.0001    0.0000   1589.6704
938.5041    0.99 …

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.0049    0.0049    0.0050    0.0050
0.0051
```




```
plot_estimates(vaf_v_005,
chain_binomial_w_beta_prior_005[:p])
```

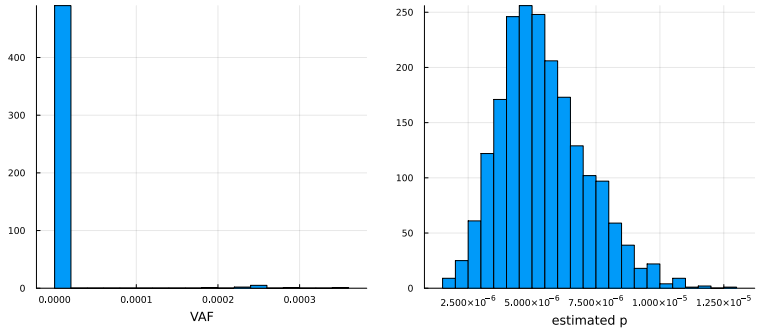
6.3 Estimate if true p is 0.00001

```
begin
    # Sampling from the posterior distribution
    @time chain_binomial_w_beta_prior_00001 =
    sample(binomial_w_beta_prior(N_v_00001, x_v_00001),
    NUTS(), 2000)
    display(chain_binomial_w_beta_prior_00001)
end
```

100%

Found initial step size
ϵ: 0.003125

```
  2.666185 seconds (1.22 M allocations: 80.550 MiB,
0.64% gc time)
Chains MCMC chain (2000×13×1 Array{Float64, 3}):

Iterations        = 1001:1:3000
Number of chains  = 1
Samples per chain = 2000
Wall duration     = 2.64 seconds
Compute duration  = 2.64 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, max_hamiltonian_energy_error, tree_depth,
numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse   ess_bulk
ess_tail      rh …
      Symbol   Float64   Float64   Float64    Float64
Float64   Float …

           p    0.0000    0.0000    0.0000   821.2807
1048.3478    1.00 …

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.0000    0.0000    0.0000    0.0000
0.0000
```




```
plot_estimates(vaf_v_00001,
chain_binomial_w_beta_prior_00001[:p])
```

7 Estimate p of NegativeBinomial(r, p) from Binomial data with different Ns

```
md"# 7 Estimate p of NegativeBinomial(r, p) from Binomial data with different Ns"
```

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

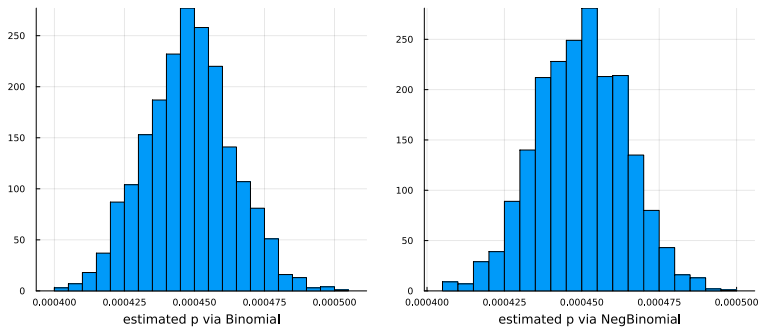
```
# Define the model
@model negbin_w_beta_prior(N, x) = begin
    # Prior distribution for the success probability
    p ~ Beta(1, 5)
    # Prior distribution for the dispersion parameter, for the NB distribution
    # r ~ Gamma(1, 1)
    #y_dist = NegativeBinomial(r, p)
    # Likelihood
    for i in eachindex(N)
        x[i] ~ NegativeBinomial(N[i]-x[i], p)
    end
end
```

```
begin
    # Sampling from the posterior distribution
    @time chain_negbin_w_beta_prior =
    sample(negbin_w_beta_prior(N_v, x_v), NUTS(), 2000)
    display(chain_negbin_w_beta_prior)
end
```

100%

Found initial step size
ϵ: 0.003125

```
  5.385602 seconds (2.58 M allocations: 177.628 Mi
B, 1.31% gc time, 38.10% compilation time)
Chains MCMC chain (2000×13×1 Array{Float64, 3}):

Iterations        = 1001:1:3000
Number of chains  = 1
Samples per chain = 2000
Wall duration     = 4.52 seconds
Compute duration  = 4.52 seconds
parameters        = p
internals         = lp, n_steps, is_accept, acceptance
_rate, log_density, hamiltonian_energy, hamiltonian_en
ergy_error, max_hamiltonian_energy_error, tree_depth,
numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse   ess_bulk
ess_tail      rh …
      Symbol   Float64   Float64   Float64    Float64
Float64   Float …

           p    0.9996    0.0000    0.0000   888.3702
1276.0136    1.00 …

2 columns omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%
97.5%
      Symbol   Float64   Float64   Float64   Float64
Float64

           p    0.9995    0.9995    0.9996    0.9996
0.9996
```




```
begin
    h_of_p_bin = histogram(chain_binomial_w_beta_prior[:p],
        xaxis="estimated p via Binomial", legend=false)
    h_of_p_negbin = histogram(-
    (chain_negbin_w_beta_prior[:p].-1),
        xaxis="estimated p via NegBinomial", legend=false)
    plot(h_of_p_bin, h_of_p_negbin, layout=(1,2),
    left_margin=5*Plots.mm,
        bottom_margin=5*Plots.mm, size=(1000,430) )
end
```

7.1 p̂ of Beta+Binomial is similar to p̂ of Beta+NegativeBinomial.

```
md"## 7.1 p̂ of Beta+Binomial is similar to p̂ of
Beta+NegativeBinomial."
```

```
" p̂ of Beta+Binomial, 0.000448 is similar to p̂ of Beta+NegativeBi
```

```
begin
    p̂₁ = median(chain_binomial_w_beta_prior[:p])
    p̂₂ = median(-(chain_negbin_w_beta_prior[:p].-1))
    " p̂ of Beta+Binomial, $(@sprintf("%0.6f", p̂₁)) is
    similar to p̂ of Beta+NegativeBinomial,
    $(@sprintf("%0.6f", p̂₂))."
end
```

8 Multinomial

```
md"# 8 Multinomial"
```

```
4×1000 Matrix{Int64}:
     4      3      5      4      0      4  …      3      4      5
    15     14     11     10     19     17         17     17     17
     6      3      4      3      6      5          5      3      5
 39975  39980  39980  39983  39975  39974      39975  39976  39973
```

```
begin
    Random.seed!(250)
    # Generate some dummy data
    data_mn = rand(Multinomial(40000, [0.0001, 0.0004,
    0.0001, 0.9994]), 1000)
end
```

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

```
# Define the model
@model function multinomial_model(data)
    no_of_alleles, no_of_samples = size(data)

    # Prior for the parameters (Dirichlet distribution)
    α = ones(no_of_alleles)  # Prior concentration
parameters

    # Parameters (probabilities of the categories)
    ϵ ~ Dirichlet(α)
    # Likelihood (multinomial distribution)
    for j in 1:no_of_samples
        data[:, j] ~ Multinomial(sum(data[:,j]), ϵ)
    end
end
```

```
"MCMCChains.Chains{Float64, AxisArrays.AxisArray{Float64, 3, Arra
```

```
begin
    # Run the model
    @time chain_mn = sample(multinomial_model(data_mn),
    NUTS(), 1_000)
    # Summarize the results
    summary(chain_mn)
end
```

```
100%
```

```
Found initial step size
ϵ: 0.000390625
```

```
  9.936392 seconds (13.98 M allocations: 1.099 GiB,
3.18% gc time, 59.30% compilation time)
```

	iteration	chain	ϵ[1]	ϵ[2]	ϵ[3]	
1	501	1	0.000100408	0.000401299	9.60591e-5	0.9
2	502	1	9.79426e-5	0.000400827	9.88822e-5	0.9
3	503	1	0.00010151	0.000408165	0.000100396	0.9
4	504	1	0.000102396	0.000406606	0.000100826	0.9
5	505	1	0.000101313	0.000402725	9.85483e-5	0.9
6	506	1	0.000102202	0.000407345	0.000101775	0.9
7	507	1	9.99532e-5	0.000405024	9.72597e-5	0.9
8	508	1	0.000103476	0.000400731	9.62322e-5	0.9
9	509	1	0.000100917	0.000397146	9.82294e-5	0.9
10	510	1	9.71677e-5	0.000411256	0.000100168	0.9

⋮ more

```
chain_mn
```

▸ (:value, :logevidence, :name_map, :info)

```
propertynames(chain_mn)
```

```
3-dimensional AxisArray{Float64,3,...} with axes:
    :iter, 501:1:1500
    :var, [Symbol("ϵ[1]"), Symbol("ϵ[2]"), Symbol("ϵ[3]"), Symbol
    :chain, 1:1
And data, a 1000×16×1 Array{Float64, 3}:
[:, :, 1] =
 0.000100408  0.000401299  9.60591e-5   …   1.2582     2.0  0.0
 9.79426e-5   0.000400827  9.88822e-5      -0.808108   2.0  0.0
 0.00010151   0.000408165  0.000100396     -0.746499   2.0  0.0
 0.000102396  0.000406606  0.000100826     -0.193956   2.0  0.0
 0.000101313  0.000402725  9.85483e-5       0.0763846  2.0  0.0
 0.000102202  0.000407345  0.000101775  …  -0.160806   2.0  0.0
 9.99532e-5   0.000405024  9.72597e-5       0.657073   2.0  0.0
 ⋮                                      ⋱
 0.000102728  0.000403545  0.000101307      0.309617   2.0  0.0
 0.000100568  0.0004038    9.85621e-5   …   0.062498   2.0  0.0
 9.87572e-5   0.000406329  0.000102453      0.61542    2.0  0.0
 0.000104567  0.00040759   9.93938e-5       0.66652    2.0  0.0
 9.89762e-5   0.000402849  0.000102203      0.21905    2.0  0.0
 0.000105195  0.000403377  0.000100183      0.67073    2.0  0.0
```

```
chain_mn.value
```

```
missing
```

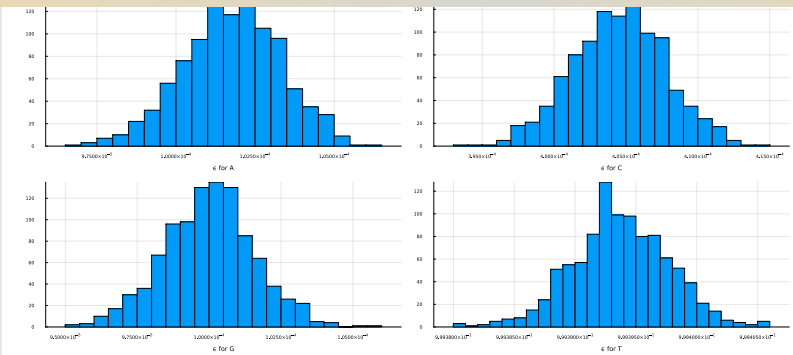
```
chain_mn.logevidence
```

▸ (varname_to_symbol = OrderedDict(ϵ[1] ⇒ Symbol("ϵ[1]"), ϵ[2] ⇒

```
chain_mn.info
```

▸ (parameters = [Symbol("ϵ[1]"), Symbol("ϵ[2]"), Symbol("ϵ[3]"), S

```
chain_mn.name_map
```

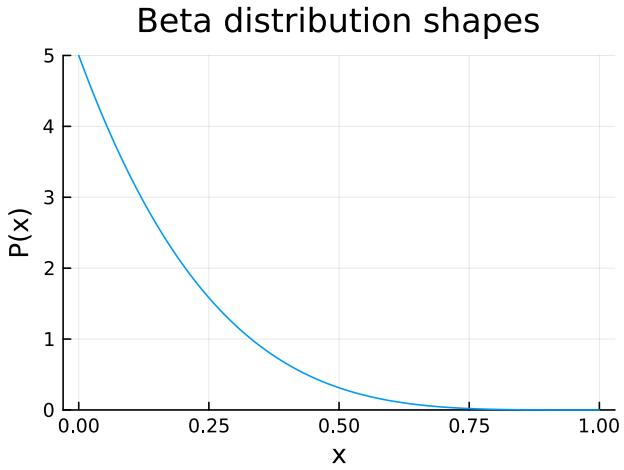



```
begin
    h_of_p1 = histogram(chain_mn[:"ϵ[1]"], xaxis="ϵ for A",
    legend=false)
    h_of_p2 = histogram(chain_mn[:"ϵ[2]"], xaxis="ϵ for C",
    legend=false)
    h_of_p3 = histogram(chain_mn[:"ϵ[3]"], xaxis="ϵ for G",
    legend=false)
    h_of_p4 = histogram(chain_mn[:"ϵ[4]"], xaxis="ϵ for T",
    legend=false)

    plot(h_of_p1, h_of_p2, h_of_p3, h_of_p4, layout=(2,2),
    left_margin=5*Plots.mm,
        bottom_margin=10*Plots.mm, size=(2000,930) )
end
```

```
using Plots

# Plot the cluster assignments over time
```

10 Visualize beta distributions

```
md"# 10 Visualize beta distributions"
```




```
plot(x -> pdf(Beta(1, 5), x),
    minimum(support(Beta(1,1))):0.01:maximum(support(Beta(1,
    1))),
    ylim=(0, 5), size=(400, 300), label=false,
    xlabel="x", ylabel="P(x)", title="Beta distribution
shapes")
```

```
0.0:0.01:1.0
```

```
0:0.01:1
```

```
#9 (generic function with 1 method)
```

```
x -> pdf(Beta(1, 5), x)
```