2nd Symposium on Advances in Approximate Bayesian Inference, 2019 1–10

AdvancedHMC.jl: A robust, modular and efficient implementation of advanced HMC algorithms

Kai Xu KAI.XU@ED.AC.UK
University of Edinburgh

Hong Ge HG344@CAM.AC.UK
University of Cambridge

Will Tebbutt WCT23@CAM.AC.UK
University of Cambridge

Mohamed Tarek M.MOHAMED@STUDENT.ADFA.EDU.AU
UNSW Canberra

Martin Trapp MARTIN.TRAPP@TUGRAZ.AT
Graz University of Technology

Zoubin Ghahramani ZOUBIN@CAM.AC.UK
University of Cambridge & Uber AI Labs

Abstract

Stan's Hamilton Monte Carlo (HMC) has demonstrated remarkable sampling robustness and efficiency in a wide range of Bayesian inference problems through carefully crafted adaption schemes to the celebrated No-U-Turn sampler (NUTS) algorithm. It is challenging to implement these adaption schemes robustly in practice, hindering wider adoption amongst practitioners who are not directly working with the Stan modelling language. AdvancedHMC.jl (AHMC) contributes a modular, well-tested, standalone implementation of NUTS that recovers and extends Stan's NUTS algorithm. AHMC is written in Julia, a modern high-level language for scientific computing, benefiting from optional hardware acceleration and interoperability with a wealth of existing software written in both Julia and other languages, such as Python. Efficacy is demonstrated empirically by comparison with Stan through a third-party Markov chain Monte Carlo benchmarking suite.

1. Introduction

Hamiltonian Monte Carlo (HMC) is an efficient Markov chain Monte Carlo (MCMC) algorithm which avoids random walks by simulating Hamiltonian dynamics to make proposals (Duane et al., 1987; Neal et al., 2011). Due to the statistical efficiency of HMC, it has been widely applied to fields including physics (Duane et al., 1987), differential equations (Kramer et al., 2014), social science (Jackman, 2009) and Bayesian inference (e.g. Bayesian neural networks; Neal, 2012). The No-U-Turn Sampler (NUTS; Hoffman and Gelman, 2014) is an extension of the HMC sampler which automatically tunes two key parameters, the leapfrog step size and integration time (aka trajectory length), which used to require manual adjustments through extensive preliminary runs. Together with a robust implementation in

the `Stan` probabilistic programming language (PPL), NUTS has become the default choice for HMC sampling for many probabilistic modelling practitioners (Carpenter et al., 2017).

Although the integration of NUTS in `Stan` makes Bayesian inference easy for domain experts relying on the `Stan` language, it is desirable to have a high quality, standalone NUTS implementation in a high-level language, e.g. for research on HMC algorithms, reproducible comparisons and real-world approximate inference applications. To this end, we introduce `AdvancedHMC.jl` (`AHMC`), a robust, modular and efficient implementation of `Stan`'s NUTS and several other commonly used HMC variants in Julia.[1]

2. A modular Hamiltonian Monte Carlo implementation

`AHMC` supports various HMC algorithms in the set below resulted from a Cartesian product of a set of HMC trajectories and a set of adaptors:

$$(\texttt{StaticTrajectory} \cup \texttt{DynamicTrajectory}) \times \texttt{Adaptor}.$$

Here `StaticTrajectory` refers to a set of HMC with fixed-length trajectory length, which contains HMC with fixed step size and step numbers and HMC with fixed total trajectory length. `DynamicTrajectory` is a set of HMC with adaptive trajectory length which is defined as a Cartesian product of four sets of different HMC components:

$$\texttt{Metric} \times \texttt{Integrator} \times \texttt{TrajectorySampler} \times \texttt{TerminationCriterion},$$

where

$$\begin{aligned}
\texttt{Metric} &= \{\texttt{UnitEuclidean}, \texttt{DiagEuclidean}, \texttt{DenseEuclidean}\} \\
\texttt{Integrator} &= \{\texttt{Leapfrog}, \texttt{JitteredLeapfrog}, \texttt{TemperedLeapfrog}\} \\
\texttt{TrajectorySampler} &= \{\texttt{Slice}, \texttt{Multinomial}\} \\
\texttt{TerminationCriterion} &= \{\texttt{ClassicNoUTurn}, \texttt{GeneralisedNoUTurn}\}
\end{aligned}$$

Finally, `Adaptor` consists of any `BaseAdaptor` or any composition of two or more `BaseAdaptor`, where $\texttt{BaseAdaptor} \in \{\texttt{Preconditioner}, \texttt{NesterovDualAveraging}\}$. A special composition called `StanHMCAdaptor` is provided to compose `Stan`'s windowed adaptor, which has been shown to be robust in practice (Carpenter et al., 2017).

2.1. Example code of building `Stan`'s NUTS using `AdvancedHMC.jl`

The code snippet below illustrates the use of `AHMC` for a given target log density function and its gradient, `logdensity_f` and `grad_f` respectively.

```
using AdvancedHMC
n_samples, n_adapts, target = 10_000, 2_000, 0.8 # set up sampling parameters
q0 = randn(D) # draw a random starting point
### Building up NUTS
metric = DiagEuclideanMetric(D) # diagonal Euclidean metric space
h = Hamiltonian(metric, logdensity_f, grad_f) # Hamiltonian on the target distribution
eps_init = find_good_eps(h, q0) # initial step size
```

1. Code is available at https://github.com/TuringLang/AdvancedHMC.jl.

```
int = Leapfrog(eps_init) # leapfrog integrator
traj = NUTS{Multinomial,GeneralisedNoUTurn}(int) # multi. sampling with gen. no U−turn
adaptor = StanHMCAdaptor( # Stan's windowed adaptor
    n_adapts, Preconditioner(metric), NesterovDualAveraging(target, eps_init)
)
samples, stats = sample(h, traj, q0, n_samples, adaptor, n_adapts) # draw samples
```

Here `logdensity_f` and `grad_f` are functions of the target distributions's log density and its gradient, which are functions independent of `AHMC` and can be, e.g. derived from different PPLs or defined by normalising flows (Rezende and Mohamed, 2015; Dinh et al., 2016; Papamakarios et al., 2017). We will show an example of such models in the next section.

2.2. GPU support for `AdvancedHMC.jl` via `CuArrays.jl`

In order to run HMC on CUDA, one only needs to change Line 3 of the demo code from `q0 = randn(D)` to `q0 = CuArray(randn(D))`, assuming `logdensity_f` and `grad_f` in Line 6 are GPU friendly, which is how `CuArrays.jl` could be used with `AdvancedHMC.jl` to run HMC on GPUs. An example using GPU accelerated `AHMC` to draw samples from a normalising flow, named `flow_model`, trained on MNIST (LeCun, 1998) is given below. The function `logpdf(m, x)` is used to compute the log density for model `m` on data batch `x`.

```
logdensity_f(x) = logpdf(flow_model, reshape(x, 784, 1))[1]
# Define gradient function via reverse AD
function grad_f(x)
    val, back = Tracker.forward(logdensity_f, x)
    grad = back(1)
    return (Tracker.data(val), Float32.(Tracker.data(grad[1][:,1])))
end
```

Here `Tracker` is an automatic differentiation (AD) library which implements reverse-mode AD. *How does it work?* All arrays in `AHMC` are abstractly typed, meaning that the concrete type is deduced at compile time from `q0`. That is to say, if `q0` is defined on the GPU, i.e. it is a `CuArray`, all internal arrays in HMC will be too.

3. Related work

A summary of the related work on HMC/NUTS implementations is given in Table 3. We want to emphasis that there exists a Python API of `AHMC` implemented by an independent team available at https://github.com/salilab/hmc.

4. Evaluations

To compare the NUTS implementation between `AHMC` and `Stan`, we consider five models from `MCMCBenchmarks.jl`, a package for benchmarking MCMC libraries in Julia.[2] The five models are 1) a *Gaussian model* (Gaussian), a simple two parameter Gaussian distribution, 2) the *signal detection model* (SDT), a model used in psychophysics and signal processing (Green et al., 1966), 3) a *linear regression* model (LR), 4) a *hierarchical Poisson regression* (HPR) model and 5) the *linear ballistic accumulator model* (LBA), a cognitive model

2. Available at https://github.com/StatisticalRethinkingJulia/MCMCBenchmarks.jl.

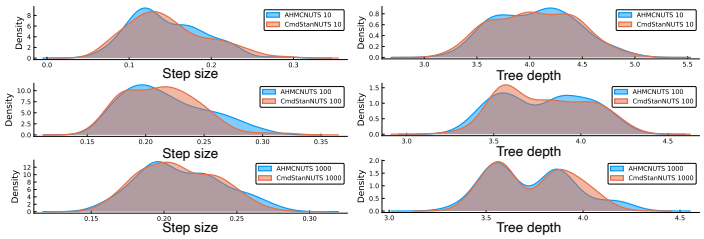
	Stan	AHMC (ours)	Pyro	TFP.MCMC	PyMC3	DynamicHMC
Adaption						
Step-size	✓	✓	✓	✓	✓	✓
Mass matrix	✓	✓	✓	✓	✓	✓
Windowed adaption	✓	✓	✓	✗	✗	✓
Hamiltonian trajectory						
Classic No-U-Turn	✓	✓	✗	✓	✗	✗
Generalised No-U-Turn	✓	✓	✓	✓	✓	✓
Fixed trajectory length	✓	✓	✓	✓	✓	✗
Trajectory sampler						
Slice	✓	✓	✓	✓	✗	✗
Multinomial	✓	✓	✓	✓	✓	✓
Integrator						
Leapfrog	✓	✓	✓	✓	✓	✓
Jittered Leapfrog	✓	✓	✗	✗	✗	✗
Tempered Leapfrog	✗	✓	✗	✗	✗	✗
GPU support	partial	✓	✓	✓	partial	partial

Table 1: Comparison of different HMC/NUTS implementations. `TFP.MCMC` refers to `TensorFlow.Probability`'s MCMC library. `DynamicHMC` is another high-quality HMC implementation in Julia. Windowed adaption is a method for joint adaption of leapfrog integration step-size and mass matrix. Windowed adaption was proposed by the Stan team (Carpenter et al., 2017), and has demonstrated remarkable robustness in a wide range of Bayesian inference problems. Partial support for GPU means the log density function can be accelerated by GPU, but the HMC sampler itself runs on CPU. Slice and Multinomial are two methods for sampling from dynamic Hamiltonian trajectories, e.g. those generated by the No-U-Turn algorithm (see e.g. Betancourt (2017) for details). Tempered leapfrog integrator improves convergence when the target distribution has multiple modes by performing tempering within Hamiltonian trajectories (Neal et al., 2011).

of perception and simple decision making (Brown and Heathcote, 2008). Please refer to Appendix A for full descriptions of these models.

4.1. Statistical property of simulated trajectories

To compare the statistical property between `Stan` and `AHMC`, we run multiple chains of NUTS with target acceptance rate 0.8 for $2,000$ steps with $1,000$ for adaptation, where the warm-up samples are dropped. Each model is benchmarked with multiple data sizes N. We compare the simulated trajectories by the distributions of step size and tree depth and the average effective sample size (ESS) for all variables. The results for the SDT model is shown in Figure 3. It can been seen that the distributions of step size and tree depth are similar and the average ESS for all variables are close, indicating `AHMC`'s NUTS is statistically similar to



	N	ESS			
		b_0	σ	b_1	b_2
`Stan`	10	413.939	266.476	381.219	423.441
`AHMC`	10	354.946	263.894	411.769	399.420
`Stan`	100	621.796	729.812	465.990	608.608
`AHMC`	100	473.005	734.189	606.996	621.543
`Stan`	1000	668.524	789.987	464.459	648.201
`AHMC`	1000	485.988	786.577	676.097	689.344

Figure 1: LR (50 runs). For the density plots, blue is for `AHMC` and orange is for `Stan` and each row is for a different size N, corresponding to the table on the right.

0	Gaussian [2]		SDT [3]		LR [2]		HPR [1]		LBA [2]	
	N	seconds	N	seconds	N	seconds	N	seconds	N	seconds
`Stan`	10	0.8039	10	0.7759	10	0.8669	10	2.4870	10	1.9179
`AHMC`	10	0.3361	10	0.3285	10	1.1356	10	19.4587	10	2.6906
`Stan`	100	0.7561	100	0.7261	100	0.9824	20	3.5025	50	7.8471
`AHMC`	100	0.3303	100	0.3201	100	1.3202	20	28.2982	50	11.0270
`Stan`	1000	0.7614	1000	0.7089	1000	2.2600	50	5.8954	200	31.3762
`AHMC`	1000	0.5081	1000	0.3179	1000	3.8326	50	40.0322	200	33.6125

Table 2: Computational efficiency for five models using [1] 25 runs, [2] 50 runs or [3] 100 runs. For `AHMC`, forward-mode AD is used for computing gradient. `AHMC` can be used together with different AD backends/packages, e.g. `ForwardDiff.jl`'s forward-mode AD, `Tracker.jl`'s reverse-mode AD and `Zygote.jl`'s source-to-source AD.

the implementation in `Stan`. Conclusions from the results of the other four models remain similar; see Appendix B for figures and tables.

4.2. Computational efficiency via running time

All the benchmark models used in this paper are implemented in `Turing` (Ge et al., 2018), a universal PPL in Julia that uses `AHMC` as its HMC backend; see Appendix C for an example `Turing` code. The average time used to run the five benchmark models for multiple times using `Stan` and using `AHMC` via `Turing` are reported in Table 2. We see that `AHMC` has comparable performance for all models except for HPR, which could be the result of the difference in the implementation of the probability mass function for the Poisson distribution. Also note that `AHMC` scales better for LBA models while `Stan` scales better for the rest of the models. This is likely due to the fact that the LBA model relies on a manually implemented distribution which is highly optimized by Julia's compiler.

5. Conclusion

We have introduced `AdvancedHMC.jl`, a new Julia package that provides robust, modular and efficient implementations of advanced HMC algorithms. We highlight the modularity of the package and compare the implemented NUTS with `Stan`, both statistically and computationally. Overall, we hope that AHMC can serve as a robust baseline for future research on HMC algorithms and can enable the development of new PPLs which take advantage of the decoupling of inference algorithms from the actual modelling language.

Acknowledgments

We would like to thank the developers of `MCMCBenchmarks.jl`, Rob Goedman and Christopher Fisher. With this package we ran all benchmarks and generated all plots presented in this paper. Since `AdvancedHMC` is an open source project, we have received many contributions from volunteers around the world. We would like to thank all contributors for their valuable work.

References

Michael Betancourt. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*, 2017.

Scott D Brown and Andrew Heathcote. The simplest complete model of choice response time: Linear ballistic accumulation. *Cognitive psychology*, 57(3):153–178, 2008.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. *arXiv preprint arXiv:1605.08803*, 2016.

Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid Monte Carlo. *Physics letters B*, 195(2):216–222, 1987.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690, Playa Blanca, Lanzarote, Canary Islands, 09–11 Apr 2018. PMLR. URL http://proceedings.mlr.press/v84/ge18b.html.

David Marvin Green, John A Swets, et al. *Signal detection theory and psychophysics*, volume 1. Wiley New York, 1966.

Matthew D Hoffman and Andrew Gelman. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1): 1593–1623, 2014.

Simon Jackman. *Bayesian analysis for the social sciences*, volume 846. John Wiley & Sons, 2009.

Andrei Kramer, Ben Calderhead, and Nicole Radde. Hamiltonian Monte Carlo methods for efficient parameter estimation in steady state dynamical systems. *BMC bioinformatics*, 15(1):253, 2014.

Yann LeCun. The MNIST database of handwritten digits. *online: http://yann.lecun.com/exdb/mnist/*, 1998.

Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

Radford M Neal et al. MCMC using Hamiltonian dynamics. *Handbook of Markov chain Monte Carlo*, 2(11):2, 2011.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pages 2338–2347, 2017.

Danilo Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1530–1538, Lille, France, 2015. PMLR.

Appendix A. Full descriptions of models used for evaluations

Below are the full descriptions of five models used for evaluations.

Gaussian Model (Gaussian) is a simple two parameter Gaussian distribution.

$$\begin{aligned}\mu &\sim \mathcal{N}(0, 1)\\ \sigma &\sim \mathcal{T}runcated(\mathcal{C}auchy(0, 5), 0, \infty)\\ y_n &\sim \mathcal{N}(\mu, \sigma)\ (n = 1, \ldots, N)\end{aligned}$$

Signal Detection Model (SDT) is a model used in psychophysics and signal processing, which decomposes performance in terms of discriminability and bias (Green et al., 1966).

$$\begin{aligned}d &\sim \mathcal{N}(0, \frac{1}{\sqrt{2}})\\ c &\sim \mathcal{N}(0, \frac{1}{\sqrt{2}})\\ x &\sim \mathrm{SDT}(d, c)\end{aligned}$$

Linear Regression Model (LR) is a linear regression with truncated Cauchy prior on the weights.

$$\begin{aligned}B_d &\sim \mathcal{N}(0, 10)\\ \sigma &\sim \mathcal{T}runcated(\mathcal{C}auchy(0, 5), 0, \infty)\\ y_n &\sim \mathcal{N}(\mu_n, \sigma)\end{aligned}$$

where $\mu = B_0 + \boldsymbol{B}^T \boldsymbol{X}, d = 1, \ldots, D$ and $n = 1, \ldots, N$.

Hierarchical Poisson Regression (HPR)

$$\begin{aligned}a_0 &\sim \mathcal{N}(0, 10)\\ a_1 &\sim \mathcal{N}(0, 1)\\ b_\sigma &\sim \mathcal{T}runcated(\mathcal{C}auchy(0, 1), 0, \infty)\\ b_d &\sim \mathcal{N}(0, b_\sigma)\\ y_n &\sim \mathcal{P}oisson(\log \lambda_n)\end{aligned}$$

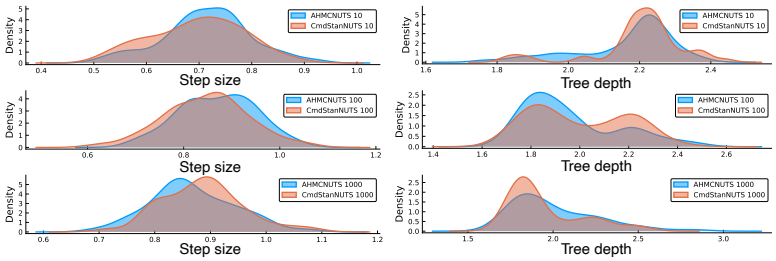
where $\log \lambda_n = a_0 + b_{z_n} + a_1 x_n, d = 1, \ldots, N_b$ and $n = 1, \ldots, N$.

Linear Ballistic Accumulator (LBA) is a cognitive model of perception and simple decision making (Brown and Heathcote, 2008).

$$\begin{aligned}\tau &\sim \mathcal{T}runcated(\mathcal{N}(0.4, 0.1), 0, mn)\\ A &\sim \mathcal{T}runcated(\mathcal{N}(0.8, 0.4), 0, \infty)\\ k &\sim \mathcal{T}runcated(\mathcal{N}(0.2, 0.3), 0, \infty)\\ \nu_d &\sim \mathcal{T}runcated(\mathcal{N}(0, 3), 0, \infty)\\ x_n &\sim \mathrm{LBA}(\boldsymbol{\nu}, \tau, A, k)\end{aligned}$$

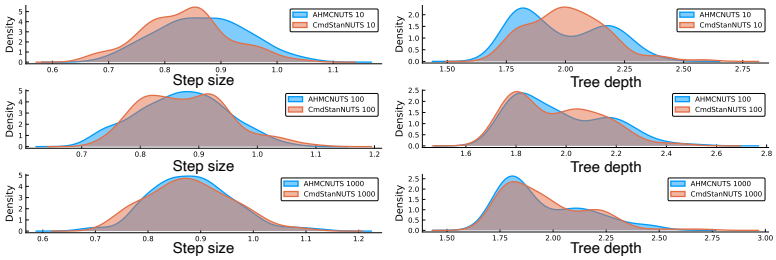
where $mn = \min_i x_{i,2}, d = 1, \ldots, N_c$ and $n = 1, \ldots, N$.

Appendix B. Statistical property of simulated trajectories



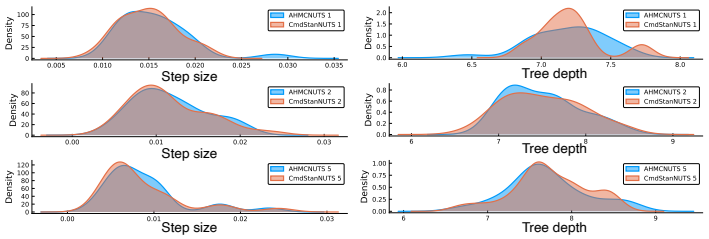
	N	ESS	
		μ	σ
`Stan`	10	513.163	466.577
`AHMC`	10	503.535	447.722
`Stan`	100	786.531	782.231
`AHMC`	100	786.531	796.628
`Stan`	1000	864.010	876.660
`AHMC`	1000	832.255	844.452

Figure 2: Gaussian (50 runs); left to right: step size, tree depth, ESS



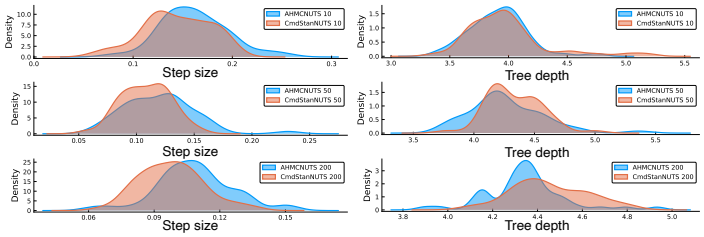
	N	ESS	
		d	c
`Stan`	10	710.762	703.327
`AHMC`	10	802.236	815.929
`Stan`	100	820.741	823.152
`AHMC`	100	814.308	846.357
`Stan`	1000	844.478	872.961
`AHMC`	1000	829.792	859.018

Figure 3: SDT (100 runs); left to right: step size, tree depth, ESS



	N	ESS		
		a_0	a_1	b_σ
`Stan`	10	221.485	215.013	266.900
`AHMC`	10	216.491	214.459	258.638
`Stan`	20	208.286	207.041	241.080
`AHMC`	20	206.458	200.469	236.546
`Stan`	50	172.484	172.982	216.586
`AHMC`	50	200.755	201.548	247.384

Figure 4: HPR (25 runs); left to right: step size, tree depth, ESS (of some variables)



	N	ESS			
		τ	A	ν_1	ν_2
`Stan`	10	226.463	282.656	305.614	276.557
`AHMC`	10	340.722	304.523	337.610	336.357
`Stan`	50	212.838	238.003	24.009	232.667
`AHMC`	50	248.249	238.979	248.331	255.421
`Stan`	200	244.926	264.967	268.793	270.36
`AHMC`	200	256.638	263.098	270.978	266.769

Figure 5: LBA (50 runs); left to right: step size, tree depth, ESS (of some variables)

Appendix C. `Turing` code for the linear regression model

Below is the code snippet of running NUTS using `Turing` for the LR model.

```
@model LR(x, y, Nd, Nc) = begin
    B ~ MvNormal(zeros(Nc), 10)
    B0 ~ Normal(0, 10)
    sigma ~ Truncated(Cauchy(0, 5), 0, Inf)
    mu = B0 .+ x * B
    y ~ MvNormal(mu, sigma)
end
x, y, Nd, Nc = ... # load data
chain = sample(LR(x, y, Nd, Nc), NUTS(2_000, 1_000, 0.8))
```