AdvancedHMC.jl: a modular implementation of Stan's no-U-turn sampler in Julia

Kai Xu[1], Hong Ge[2], Will Tebbutt[2], Mohamed Tarek[3], Martin Trapp[4] and Zoubin Ghahramani[2,5]

[1]University of Edinburgh [2]University of Cambridge [3]UNSW Canberra [4]Graz University of Technology [5]Uber AI Labs

Abstract

The No-U-Turn Sampler (NUTS) in `Stan` (Hoffman and Gelman, 2014; Carpenter et al., 2017) has demonstrated remarkable sampling robustness and efficiency in a wide range of Bayesian inference problems, due to the use of dynamic Hamiltonian trajectory, and a fine-tuned joint adaptation of step-size and mass matrix. Motivated by these successes, we present `AdvancedHMC.jl` (`AHMC`), a pure Julia implementation of `Stan`'s built-in NUTS algorithm and related adaptation methods. We hope `AdvancedHMC.jl` can help expose `Stan`'s NUTS to a wider range of users, e.g. those who want to write their models by hand, or using a different probabilistic programming language (e.g. `Turing`, `Soss`). In our package, NUTS is defined as a combination of individual components with abstractions partially inspired by (Betancourt, 2017).

Hamiltonian Monte Carlo Components

Hamiltonian Monte Carlo (HMC) simulates Hamiltonian dynamics to make proposals for a Markov chain (Neal et al., 2011). `AdvancedHMC.jl` supports various HMC samplers below.

$$(\texttt{StaticHMC} \cup \texttt{DynamicHMC}) \times \texttt{Adaptor}.$$

Here `StaticHMC` are HMC with fixed-length trajectories and `DynamicHMC` are HMC with adaptive trajectory length which can be created by composing NUTS components as follows

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

where

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

`Adaptor` can be composed from base adaptors

$$\texttt{BaseAdaptor} \in \{\texttt{Preconditioner}, \texttt{NesterovDualAveraging}\}.$$

Note 1: `Preconditioner` behaves differently based on the choice of metric spaces.
Note 2: `StanHMCAdaptor`, a specific composition of base adaptors that is equivalent to `Stan`'s windowed adaptor, is provided. This adaptor has been proved to be robust in practice.

Benchmark Models

We use five models from `MCMCBenchmarks.jl` to compare NUTS between `AdvancedHMC.jl` and `Stan`.

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

$$\mu \sim \mathcal{N}(0,1), \quad \sigma \sim \mathcal{Truncated}(\mathcal{Cauchy}(0,5), 0, \infty), \quad y_n \sim \mathcal{N}(\mu, \sigma) \ (n = 1, \ldots, N)$$

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

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

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

$$B_d \sim \mathcal{N}(0,10), \ \sigma \sim \mathcal{Truncated}(\mathcal{Cauchy}(0,5), 0, \infty), \ y_n \sim \mathcal{N}(\mu_n, \sigma),$$

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

Hierarchical Poisson Regression (HPR)

$$a_0 \sim \mathcal{N}(0,10), \ a_1 \sim \mathcal{N}(0,1), \ b_\sigma \sim \mathcal{Truncated}(\mathcal{Cauchy}(0,1), 0, \infty), \ b_d \sim \mathcal{N}(0, b_\sigma), \ y_n \sim \mathcal{Poi}(\log \lambda_n),$$

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.

$$\tau \sim \mathcal{Truncated}(\mathcal{N}(0.4, 0.1), 0, mn), \quad A \sim \mathcal{Truncated}(\mathcal{N}(0.8, 0.4), 0, \infty),$$
$$k \sim \mathcal{Truncated}(\mathcal{N}(0.2, 0.3), 0, \infty), \quad \nu_d \sim \mathcal{Truncated}(\mathcal{N}(0,3), 0, \infty), \quad x_n \sim \mathrm{LBA}(\nu, \tau, A, k)$$

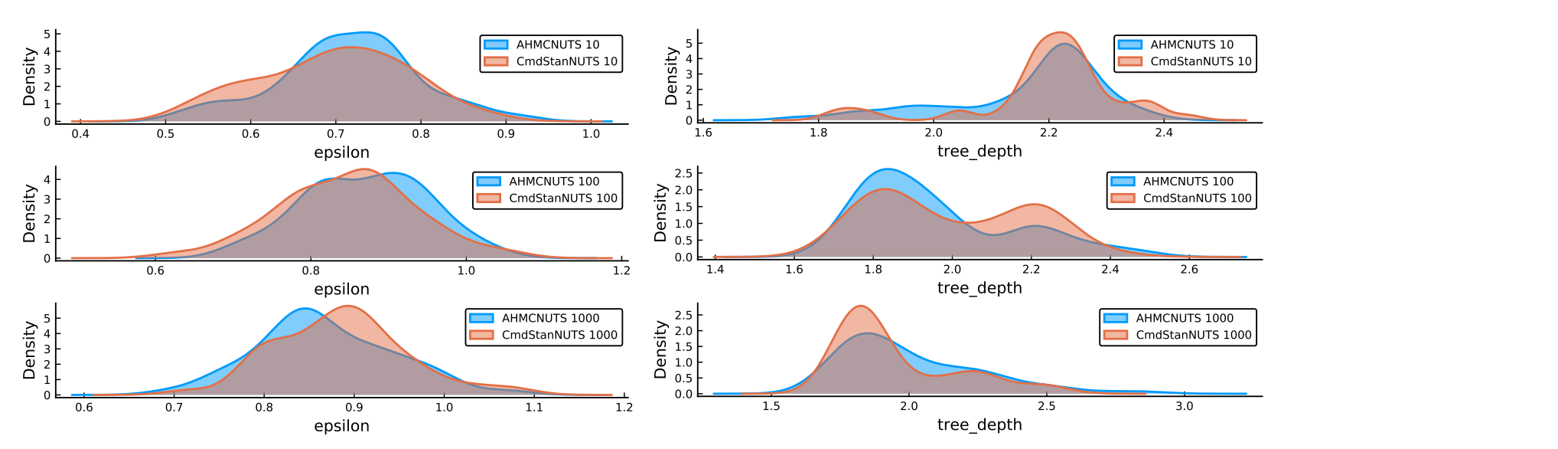
where $mn = \min_i x_{i,2}, d = 1, \ldots, N_c$ and $n = 1, \ldots, N$.

Example Code of Building Stan's NUTS using AHMC

```julia
using AdvancedHMC
n_samples, n_adapts, target = 10_000, 2_000, 0.8 # set up sampling parameters
q_init = 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, q_init) # initial step size
int = Leapfrog(eps_init) # leapfrog integrator
traj = NUTS{Multinomial,GeneralisedNoUTurn}(int) # multinomial sampling with generalised no U−turn
adaptor = StanHMCAdaptor( # Stan's windowed adaptor
    n_adapts, Preconditioner(metric), NesterovDualAveraging(target, eps_init)
)
samples, stats = sample(h, traj, q_init, n_samples, adaptor, n_adapts) # draw samples
```

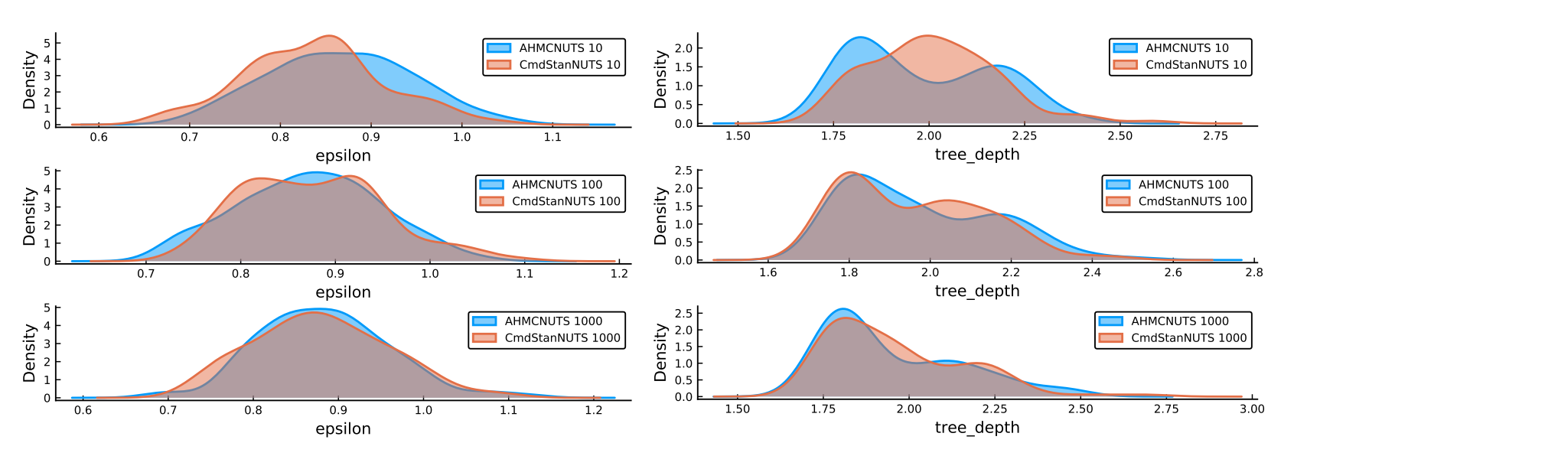
Sampling Efficiency: Stan's NUTS v.s. AHMC

To compare the sampling efficiency between `Stan` and `AHMC`, we run multiple runs of NUTS with target acceptance rate 0.8 for 2,000 runs with 1,000 adaptation steps, where the warm-up samples dropped. Below are figures of distributions of step size and tree depth, and the mean effective sample size (ESS) for different variables.



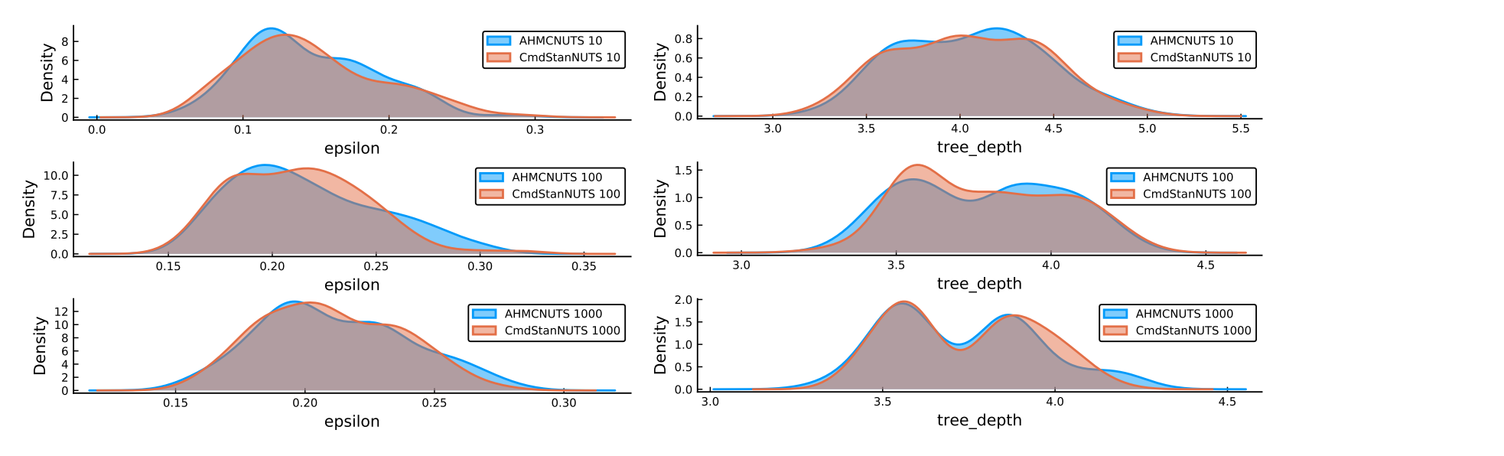
	N	ESS μ	ESS σ
`Stan`	10	513.163	466.577
`AHMC`	10	503.535	447.722
`Stan`	100	786.531	782.231
`AHMC`	100	786.531	796.628
`Stan`	1000	864.010	876.660
`AHMC`	1000	832.255	844.452

Fig. 1: Gaussian (50 runs); left to right: step size, tree depth, ESS



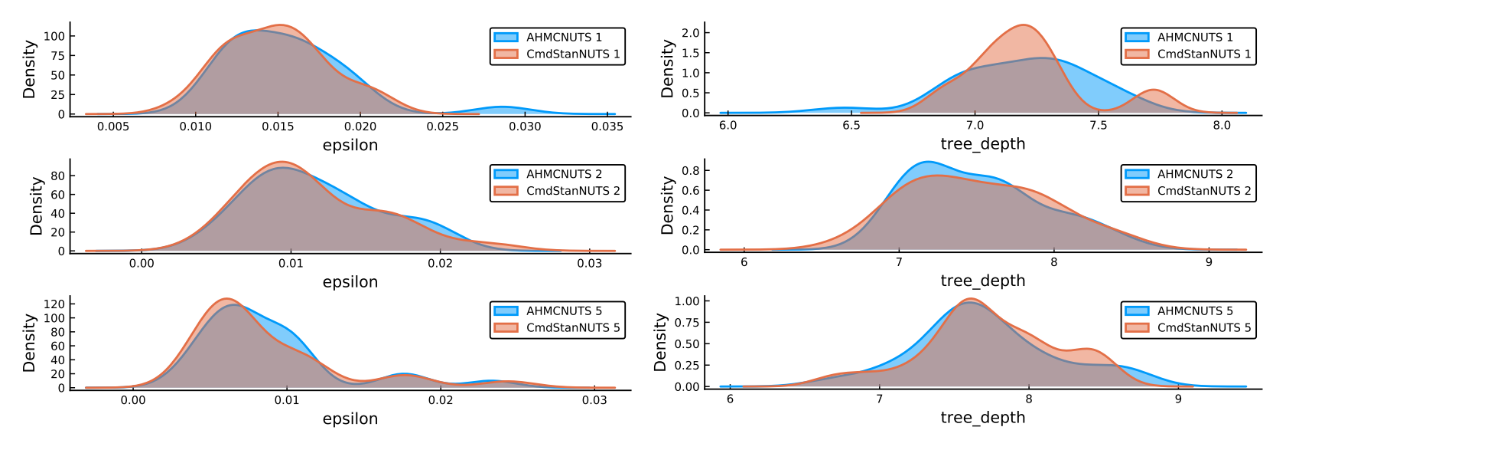
	N	ESS d	ESS c
`Stan`	10	710.762	703.327
`AHMC`	10	802.236	815.929
`Stan`	100	820.741	823.152
`AHMC`	100	814.308	846.357
`Stan`	1000	844.478	872.961
`AHMC`	1000	829.792	859.018

Fig. 2: SDT (100 runs); left to right: step size, tree depth, ESS



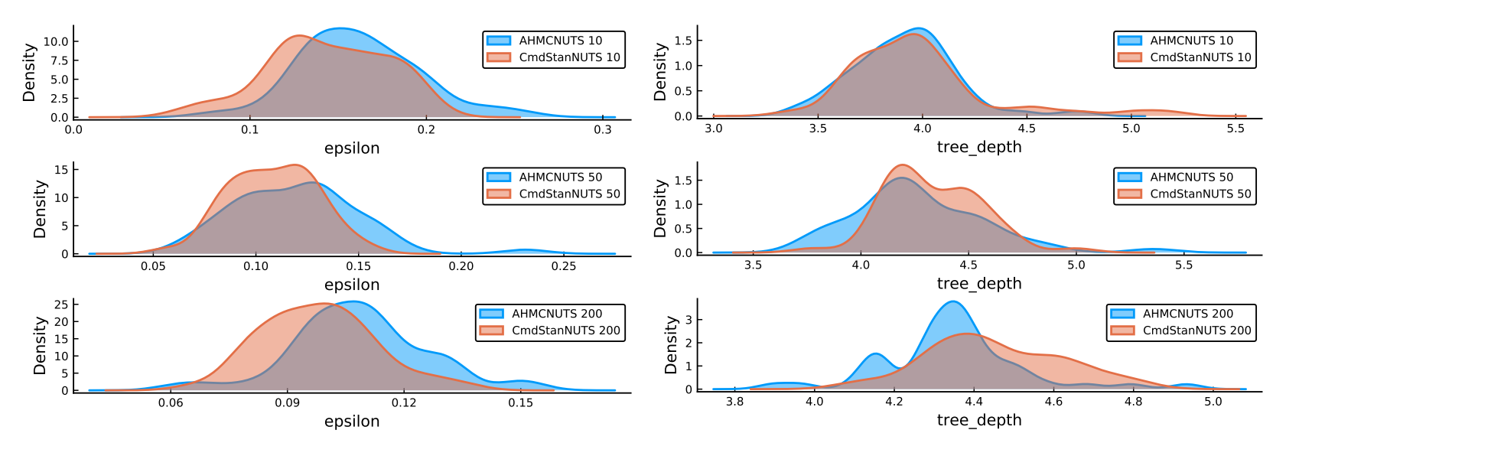
	N	ESS b_0	ESS σ	ESS b_1	ESS b_2
`Stan`	10	413.939	266.476	381.219	423.441
`AHMC`	10	354.946	263.894	411.769	399.420
`Stan`	100	621.796	729.812	465.990	608.608
`AHMC`	100	473.005	734.189	606.996	621.543
`Stan`	1000	668.524	789.987	464.459	648.201
`AHMC`	1000	485.988	786.577	676.097	689.344

Fig. 3: LR (50 runs); left to right: step size, tree depth, ESS



	N	ESS a_0	ESS a_1	ESS 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

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



	N	ESS τ	ESS A	ESS ν_1	ESS 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	235.009	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

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

Computational Efficiency: Stan v.s. Turing

`Turing.jl` is a probabilistic programming language (PPL) in Julia that uses `AdvancedHMC.jl` as its HMC backend. All the benchmark models are written in `Turing` and `AdvancedHMC.jl` is called by `Turing.jl` to run the NUTS. Below is an example of running NUTS on the LR model using `Turing`.

```julia
@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))
```

The time to run the five benchmark models in `Stan` and `Turing` are reported in the table below.

	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. 1: Time comparisons between `Stan` and `Turing` (`AHMC`) for five models using [1] 25 runs, [2] 50 runs or [3] 100 runs.

Easy Integration of Other Julia Packages

`Bijectors.jl` is used inside `Turing.jl` to do automatic transformations of constrained variables to run HMC. E.g. a random variable from $\mathcal{Truncated}(\mathcal{Cauchy}(0,5), 0, \infty)$ is constrained to be positive and will be transformed to the real space by the log function automatically.

`CuArrays.jl` could be used with `AdvancedHMC.jl` to run NUTS on GPUs. In order to run NUTS using CUDA, one only needs to change Line 3 of the demo code from `q_init = randn(D)` to `q_init = CuArray(randn(D))`, assuming `logdensity_f` and `grad_f` in Line 6 are GPU friendly; if it is written in pure Julia, it probably supports GPUs acceleration automatically.
How does it work? All arrays in `AdvancedHMC.jl` are abstractly typed, meaning that the concrete type is deduced at compile time from `q_init`. That is to say, if `q_init` is on the GPU i.e. is a CuArray, all the internal arrays in the NUTS will be too.

`SoSS.jl` is another PPL in Julia that uses `AdvancedHMC.jl` as its backend. It is easy for PPLs in Julia with different domain specific languages (DSLs) to use the HMC implementation in `AdvancedHMC.jl`.

`DifferentialEquations.jl` is the state-of-the-art numerical differential equations solver package, implemented in pure Julia. As such, its solvers can be employed in Turing models, thus enabling `AHMC` to perform Bayesian inference in the parameters of differential equation models.

`Flux.jl` is a deep learning packages in Julia. Neural models defined by `Flux.jl` can be directly used in Turing models. E.g. one can implement a Bayesian neural network in Turing by defining priors on the weights of a Flux-based neural network, and NUTS can be used to draw samples of the weights.

Acknowledgements

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 poster.

References

Betancourt, M. (2017). A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*.
Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., … Riddell, A. (2017). Stan: a probabilistic programming language. *Journal of statistical software*, *76*(1).
Hoffman, M. D. & Gelman, A. (2014). The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, *15*(1), 1593–1623.
Neal, R. M. et al. (2011). MCMC using Hamiltonian dynamics. *Handbook of Markov chain Monte Carlo*, *2*(11), 2.