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

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]Google AI

Abstract

`Stan`'s Hamiltonian 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 on CPU is demonstrated empirically by comparison with `Stan` and efficacy with vectorized HMC on both CPU and GPU is compared with `TensorFlow Probability`. `AdvancedHMC.jl` is available at https://github.com/TuringLang/AdvancedHMC.jl.

Hamiltonian Monte Carlo Components

`AHMC` supports a wide range of 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 by four sets of different HMC components:

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

$$\text{where} \quad \begin{aligned} \texttt{Metric} &= \{\texttt{UnitEuclidean}, \texttt{DiagEuclidean}, \texttt{DenseEuclidean}\} \\ \texttt{Integrator} &= \{\texttt{Leapfrog}, \texttt{JitteredLeapfrog}, \texttt{TemperedLeapfrog}, \texttt{DiffEqIntegrator}\} \\ \texttt{TrajectorySampler} &= \{\texttt{SliceTS}, \texttt{MultinomialTS}\} \\ \texttt{TerminationCriterion} &= \{\texttt{ClassicNoUTurn}, \texttt{GeneralisedNoUTurn}, \texttt{StrictGeneralisedNoUTurn}\} \end{aligned}$$

`DiffEqIntegrator` includes > 30 ordinary differential equation (ODE) integrators from `DifferentialEquations.jl`. Finally, `Adaptor` consists of any `BaseAdaptor` or any composition of two or more `BaseAdaptor`, where `BaseAdaptor` $\in$ {`Preconditioner`, `NesterovDualAveraging`}. A special composition called `StanHMCAdaptor` is provided to compose `Stan`'s windowed adaptation, which has been shown to be robust in practice (Carpenter et al., 2017).

Example Code of Building `Stan`'s NUTS using `AHMC`

The code snippet below illustrates how to construct NUTS with `AHMC`. `logdensity_f`.

```julia
1  using AdvancedHMC, Distributions, ForwardDiff
2
3  # Choose parameter dimensionality and initial parameter value
4  D = 10; initial_theta = rand(D)
5
6  # Define the target distribution
7  logprob(theta) = logpdf(MvNormal(zeros(D), ones(D)), theta)
8
9  # Set the number of samples to draw and warmup iterations
10 n_samples, n_adapts = 2_000, 1_000
11
12 # Define a Hamiltonian system
13 metric = DiagEuclideanMetric(D)
14 hamiltonian = Hamiltonian(metric, logprob, ForwardDiff)
```

```julia
1  # Define a leapfrog solver, with initial step size chosen heuristically
2  integrator = Leapfrog(find_good_stepsize(hamiltonian, initial_theta))
3
4  # Define an HMC sampler, with the following components
5  # — multinomial sampling scheme,
6  # — generalised No—U—Turn criteria, and
7  # — windowed adaption for step—size and diagonal mass matrix
8  proposal = NUTS{MultinomialTS, GeneralisedNoUTurn}(integrator)
9  adaptor = StanHMCAdaptor(MassMatrixAdaptor(metric), StepSizeAdaptor(0.8, integrator))
10
11 # Run the sampler to draw samples from the specified Gaussian, where
12 # — `samples` will store the samples
13 # — `stats` will store diagnostic statistics for each sample
14 samples, stats = sample(hamiltonian, proposal, initial_theta, n_samples, adaptor, n_adapts; progress=true)
```

Benchmark Models

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

Gaussian Model (Gaussian): $\mu \sim \mathcal{N}(0, 1)$, $\sigma \sim \mathcal{Truncated}(\mathcal{Cauchy}(0, 5), 0, \infty)$, $y_n \sim \mathcal{N}(\mu, \sigma)$ $(n = 1, \ldots, N)$
Signal Detection Model (SDT), a model used in psychophysics and signal processing: $d \sim \mathcal{N}(0, \frac{1}{\sqrt{2}})$, $c \sim \mathcal{N}(0, \frac{1}{\sqrt{2}})$, $x \sim \mathrm{SDT}(d, c)$
Linear Regression Model (LR) 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), a cognitive model of perception and simple decision making: $\tau \sim \mathcal{Truncated}(\mathcal{N}(0.4, 0.1), 0, mn)$, $A \sim \mathcal{Truncated}(\mathcal{N}(0.8, 0.4), 0, \infty)$, $k \sim \mathcal{Truncated}(\mathcal{N}(0.2, 0.3), 0, \infty)$, $\nu_d \sim \mathcal{Truncated}(\mathcal{N}(0, 3), 0, \infty)$, $x_n \sim \mathrm{LBA}(\nu, \tau, A, k)$ where $mn = \min_i x_{i,2}, d = 1, \ldots, N_c$ and $n = 1, \ldots, N$.

All benchmark models are written in `Turing` (Ge et al., 2018), a probabilistic programming language in Julia which uses `AdvancedHMC.jl` as its HMC backend.

NUTS Implementation: `Stan` v.s. `Turing`

	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.

Vectorized HMC on CPU and GPU: `TensorFlow Probability` v.s. `AHMC`

Simply changing `init_theta = rand(D)` to `init_theta = rand(D, n_chains)` in Line 4 of our example code makes HMC runs in vectorized mode; wrapping `init_theta` with `CuArray` moves the all the computation to GPU with the support of `CUDA.jl`.

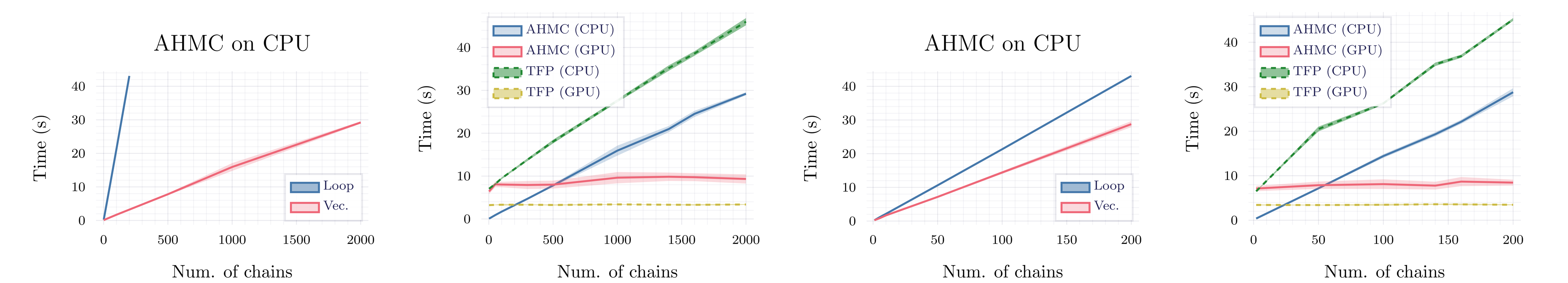

Fig. 1: Time to draw 2,000 samples using HMC with step size 0.2 and step number 5 from a multivariate standard Gaussian; left 2: 50D and right 2: 500D

Related Work

		`Stan`	`AHMC` (ours)	`Pyro`	`TensorFlow Probability`	`PyMC3`	`DynamicHMC`
Adaption	Step-size	✓	✓	✓	✓	✓	✓
	Mass matrix	✓	✓	✓	✓	✓	✓
	Windowed adaption	✓	✓	✗	✗	✗	✓
Trajectory	Classic No-U-Turn	✓	✓	✗	✓	✗	✗
	Generalised No-U-Turn	✓	✓	✓	✓	✓	✓
	Strict Generalised No-U-Turn	✓	✓	✓	✓	✓	✗
	Fixed length	✓	✓	✓	✓	✓	✗
Trajectory sampler	Slice	✓	✓	✓	✓	✗	✗
	Multinomial	✓	✓	✓	✓	✓	✓
Integrator	Leapfrog	✓	✓	✓	✓	✓	✓
	Jittered Leapfrog	✓	✓	✗	✗	✗	✗
	Tempered Leapfrog	✗	✓	✗	✗	✗	✗
	General ODE Integrator	✗	✓	✗	✗	✗	✗
	GPU support	partial	✓	✓	✓	partial	partial

Table. 2: Comparison of different HMC frameworks. `DynamicHMC` is another high-quality HMC implementation in Julia. 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 Hamiltonian trajectories (Betancourt, 2017). Tempered leapfrog improves convergence for multi-mode targets by performing tempering (Neal et al., 2011).

Acknowledgements

We would like to thank the developers of `MCMCBenchmarks.jl`, Rob Goedman and Christopher Fisher, for their benchmark framework to compare `AHMC` against Stan. We also would like to thank Christopher Rackauckas for making `AHMC` support a wide range of integrators from `DifferentialEquations.jl`.

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., Brubaker, M., Guo, J., Li, P., & Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software, 76*(1).
Ge, H., Xu, K., & Ghahramani, Z. (2018). Turing: A language for flexible probabilistic inference. *AISTATS*, 1682–1690. http://proceedings.mlr.press/v84/ge18b.html
Neal, R. M. et al. (2011). MCMC using Hamiltonian dynamics. *Handbook of Markov chain Monte Carlo, 2*(11), 2.