AdvancedHMC.jl: A robust, modular and efficient implementation of advanced HMC algorithms
Kai Xu1, Hong Ge2, Will Tebbutt2, Mohamed Tarek3, Martin Trapp4 and Zoubin Ghahramani2,5
1University of Edinburgh  2University of Cambridge  3UNSW Canberra  4Cran University of Technology  5Google 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 adaptors to the celebrated No-U-Turn sampler (NUTS) algorithm. It is challenging to implement these adaptors schemes robustly in practice, hindering wider adoption amongst practitioners who are not directly working with the Stan modelling language. AdvancedHMC.jl (AHMC) contributes to a modular, well-tested, stand-alone implementation of NUTS that covers and extends Stan’s NUTS algorithm. AHMC is 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 vectored 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:

- (StaticTrajectory ∪ DynamicTrajectory) ∪ Adaptor

Here, StaticTrajectory refers to a set of HMC with fixed-length trajectory length, which combine 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:

- Metric × Integrator × TrajectorySampler × TerminationCriterion

where

- Metric = UnitSizedDiagonal, DenseDiagonal
- Integrator = Leapfrog, Zitterbewegung, TemporalLeapfrog
- TrajectorySampler = SliceTS, MultinomialTS
- TerminationCriterion = ClassicalTurn, GeneralisedNoUTurn, StrictGeneralisedNoUTurn

DiffIntegrator includes a 30 Ordinary Differential Equation (ODE) integrators from DifferentialEquations.jl. Finally, Adaptor consists of any BaseAdaptor or any composition of two or more BaseAdaptors, where BaseAdaptor ∈ {Prec-conditioner, NesterovMalaBalancer}. A special composition StanHMCAdaptor is provided to compose Stan’s adapted version, 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.f

```julia
using AdvancedHMC, Distributions, ForwardDiff
D = 10; initial_theta = rand(D)

# Define a leapfrog solver, with initial step size chosen heuristically
integrator = Leapfrog(find_good_stepsize(hamiltonian, initial_theta))

# Define an HMC sampler, with the following components
# - multistep sampling scheme
# - generalized No-U-Turn criteria, and
# - windowed adaptation for step-size and diagonal mass matrix
# proposal = NUTSMultinomialTS(GeneralisedNoUTurn(integrator))
# adaptor = StanHMCAdaptor(HamiltonianMalaAdaptor(integrator, StanHMCAdaptor(R(K, integrator))))

# Run the sampler to draw samples from the specified Gaussian, where
# - samples will store the samples
# - stats will store diagnostic statistics for each sample
# samples, stats = sample(hamiltonian, proposal, initial_theta, n_samples, n_adapt = 2000, 1000)
```

Benchmark Models

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

<table>
<thead>
<tr>
<th>Gaussian Model (Gaussian)</th>
<th>Signal Detection Model (SDT)</th>
<th>Hierarchical Poisson Regression (HPR)</th>
<th>Linear Ballistic Accumulator (LBA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>μ ~ N(0,1), σ ~ TruncatedCauchy(0, 5, 0, 0), y ~ N(μ, σ) (n = 1, ..., N)</td>
<td>N ~ SDT(d)</td>
<td>log λ ~ N(0, 1), a0 ~ N(0, 10), b0 ~ TruncatedCauchy(0, 1, 0, 0), k ~ N(0, 1), y ~ Pois(kλ), where log λ = a0 + b0 * x + 1, N = 1, ..., N</td>
<td></td>
</tr>
</tbody>
</table>

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.

Vectorized HMC on CPU and GPU: TensorFlow Probability vs. 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 Catarray moves all the computation to CPU with the support of CUDA.jl.

![Graph](image1.png)

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 = 50D and right = 500D

Table: 2 Comparison of different HMC frameworks. DynamicHMC is another high-quality HMC implementation in Julia. Partial support for GPU means that the log density function can be accelerated by GPU, but the HMC sampler itself runs on CPU. Site and Multinomial are two methods for sampling from Hamiltonian trajectories (Betancourt, 2017). Temporal leapfrog improves convergence for multi-mode targets by performing tempering (Neal et al., 2011).

<table>
<thead>
<tr>
<th>HMC Framework</th>
<th>Site</th>
<th>Multinomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicHMC</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Stan (CPU)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Stan (GPU)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>AHMC (CPU)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>AHMC (GPU)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>TensorFlow P3</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table: Comparison of different HMC frameworks. DynamicHMC is another high-quality HMC implementation in Julia. Partial support for GPU means that the log density function can be accelerated by GPU, but the HMC sampler itself runs on CPU. Site and Multinomial are two methods for sampling from Hamiltonian trajectories (Betancourt, 2017). Temporal 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, Bob Goodman and Christopher Fasel, for their benchmark framework to compare AHMC against Stan. We would also like to thank Christopher Rackauckas for making AHMC support a wide range of integrators from DifferentialEquations.jl.

References