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

Kai Xu
University of Edinburgh
kai.xu@ed.ac.uk

Hong Ge
University of Cambridge
hg344@cam.ac.uk

Will Tebbutt
University of Cambridge
wct23@cam.ac.uk

Mohamed Tarek
UNSW Canberra
m.mohamed@student.adfa.edu.au

Martin Trapp
Graz University of Technology
martin.trapp@tugraz.at

Zoubin Ghahramani
University of Cambridge & Uber AI Labs
zoubin@cam.ac.uk

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 black-box nature of Stan’s NUTS makes Bayesian inference easy for domain experts relying on Stan, it is desirable to have a high quality 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 HMC library

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:

\[(\text{StaticTrajectory} \cup \text{DynamicTrajectory}) \times \text{Adaptor}.\]

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:

\[\text{Metric} \times \text{Integrator} \times \text{TrajectorySampler} \times \text{TerminationCriterion},\]

where

\[
\text{Metric} = \{\text{UnitEuclidean}, \text{DiagEuclidean}, \text{DenseEuclidean}\}
\]
\[
\text{Integrator} = \{\text{Leapfrog}, \text{JitteredLeapfrog}, \text{TemperedLeapfrog}\}
\]
\[
\text{TrajectorySampler} = \{\text{Slice}, \text{Multinomial}\}
\]
\[
\text{TerminationCriterion} = \{\text{ClassicNoUTurn}, \text{GeneralisedNoUTurn}\}
\]

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 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.

```
1 using AdvancedHMC
2 n_samples, n_adapts, target = 10_000, 2_000, 0.8 # set up sampling parameters
3 q0 = randn(D) # draw a random starting point
4
5 ### Building up NUTS
6 metric = DiagEuclideanMetric(D) # diagonal Euclidean metric space
7 h = Hamiltonian(metric, logdensity_f, grad_f) # Hamiltonian on the target distribution
8 eps_init = find_good_eps(h, q0) # initial step size
```

¹ Code is available at https://anonymous.4open.science/r/27c8d4a6-8ee3-452a-8a63-db8fb3408182/.
AdvancedHMC

```julia
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’ 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`.

```julia
logdensity_f(x) = logpdf(flow_model, reshape(x, 784, 1))[1]
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 A in the appendix. We want to emphasis that there exists a Python API of AHMC implemented by an independent team available at [https://github.com/salilab/hmc](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. 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
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.

<table>
<thead>
<tr>
<th></th>
<th>Stan 10</th>
<th>413.939</th>
<th>266.476</th>
<th>381.219</th>
<th>423.441</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AHMC 10</td>
<td>354.946</td>
<td>263.894</td>
<td>411.769</td>
<td>399.420</td>
</tr>
<tr>
<td></td>
<td>Stan 100</td>
<td>621.796</td>
<td>729.812</td>
<td>466.990</td>
<td>608.608</td>
</tr>
<tr>
<td></td>
<td>AHMC 100</td>
<td>473.005</td>
<td>731.189</td>
<td>606.986</td>
<td>621.543</td>
</tr>
<tr>
<td></td>
<td>Stan 1000</td>
<td>668.524</td>
<td>789.987</td>
<td>464.459</td>
<td>648.201</td>
</tr>
<tr>
<td></td>
<td>AHMC 1000</td>
<td>485.988</td>
<td>786.577</td>
<td>676.097</td>
<td>689.344</td>
</tr>
</tbody>
</table>

Table 1: Computational efficiency for five models using 125 runs, 250 runs or 3100 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.

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 the implementation in Stan. Conclusions from the results of the other four models remain similar; see Appendix C 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 D 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 1. 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 distribu-
tion. 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.

References


### Appendix A. Related Work

<table>
<thead>
<tr>
<th>Adaption</th>
<th>Stan</th>
<th>AHMC (ours)</th>
<th>Pyro</th>
<th>TFP.MCMC</th>
<th>PyMC3</th>
<th>DynamicHMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step-size</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mass matrix</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Windowed adaption</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hamiltonian trajectory</th>
<th>Stan</th>
<th>AHMC (ours)</th>
<th>Pyro</th>
<th>TFP.MCMC</th>
<th>PyMC3</th>
<th>DynamicHMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic No-U-Turn</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Generalised No-U-Turn</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fixed trajectory length</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trajectory sampler</th>
<th>Stan</th>
<th>AHMC (ours)</th>
<th>Pyro</th>
<th>TFP.MCMC</th>
<th>PyMC3</th>
<th>DynamicHMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Multinomial</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Integrator</th>
<th>Stan</th>
<th>AHMC (ours)</th>
<th>Pyro</th>
<th>TFP.MCMC</th>
<th>PyMC3</th>
<th>DynamicHMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leapfrog</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jittered Leapfrog</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Tempered Leapfrog</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU support</th>
<th>Stan</th>
<th>AHMC (ours)</th>
<th>Pyro</th>
<th>TFP.MCMC</th>
<th>PyMC3</th>
<th>DynamicHMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>partial</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>

Table 2: 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**).

### Appendix B. 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.

\[
\begin{align*}
\mu & \sim \mathcal{N}(0, 1) \\
\sigma & \sim \text{Truncated}(\text{Cauchy}(0, 5), 0, \infty) \\
y_n & \sim \mathcal{N}(\mu, \sigma) \ (n = 1, \ldots, N)
\end{align*}
\]
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).

\[
\begin{align*}
  d &\sim \mathcal{N}(0, \frac{1}{\sqrt{2}}) \\
  c &\sim \mathcal{N}(0, \frac{1}{\sqrt{2}}) \\
  x &\sim \text{SDT}(d, c)
\end{align*}
\]

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

\[
\begin{align*}
  B_d &\sim \mathcal{N}(0, 10) \\
  \sigma &\sim \text{Truncated}(\text{Cauchy}(0, 5), 0, \infty) \\
  y_n &\sim \mathcal{N}(\mu_n, \sigma)
\end{align*}
\]

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

Hierarchical Poisson Regression (HPR)

\[
\begin{align*}
  a_0 &\sim \mathcal{N}(0, 10) \\
  a_1 &\sim \mathcal{N}(0, 1) \\
  b_\sigma &\sim \text{Truncated}(\text{Cauchy}(0, 1), 0, \infty) \\
  b_d &\sim \mathcal{N}(0, b_\sigma) \\
  y_n &\sim \text{Poisson}(\log \lambda_n)
\end{align*}
\]

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

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

\[
\begin{align*}
  \tau &\sim \text{Truncated}(\mathcal{N}(0.4, 0.1), 0, mn) \\
  A &\sim \text{Truncated}(\mathcal{N}(0.8, 0.4), 0, \infty) \\
  k &\sim \text{Truncated}(\mathcal{N}(0.2, 0.3), 0, \infty) \\
  \nu_d &\sim \text{Truncated}(\mathcal{N}(0, 3), 0, \infty) \\
  x_n &\sim \text{LBA}(\nu, \tau, A, k)
\end{align*}
\]

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

Appendix C. Statistical property of simulated trajectories

Appendix D. Turing code for the linear regression model

Below is the code snippet of running NUTS using Turing for the LR model.
Table 1: Comparison of NUTS and AHMC for different sample sizes.  

<table>
<thead>
<tr>
<th>Method</th>
<th>Sample Size</th>
<th>( \mu )</th>
<th>( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stan</td>
<td>10</td>
<td>513.163</td>
<td>466.577</td>
</tr>
<tr>
<td>AHMC</td>
<td>10</td>
<td>503.535</td>
<td>447.722</td>
</tr>
<tr>
<td>Stan</td>
<td>100</td>
<td>786.531</td>
<td>782.231</td>
</tr>
<tr>
<td>AHMC</td>
<td>100</td>
<td>786.531</td>
<td>796.628</td>
</tr>
<tr>
<td>Stan</td>
<td>1000</td>
<td>864.010</td>
<td>876.660</td>
</tr>
<tr>
<td>AHMC</td>
<td>1000</td>
<td>832.255</td>
<td>844.452</td>
</tr>
</tbody>
</table>

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

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

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

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

```r
@model LR(x, y, N, d, 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))