~~~
---
title: "HMC Learn"
output: pdf_document
vignette: >
  %\VignetteIndexEntry{HMC_learn}
  %\VignetteEngine{knitr::rmarkdown}
  %\VignetteEncoding{UTF-8}
header-includes:
  - \usepackage{setspace}
  - \doublespacing
  - \usepackage{amsmath}
  - \usepackage{algorithm}
  - \usepackage{algpseudocode}
  - \usepackage{enumitem}
  - \usepackage{bm}
---

```{r, include = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>"
)
```

```{r setup}
library(hmclearn)
```
~~~

Introduction

Hamiltonian Monte Carlo (HMC) has emerged as a general purpose tool for Bayesian practitioners. A key advantage of HMC over more traditional Markov Chain Monte Carlo (MCMC) algorithms is its improved computational efficiency in fitting high-dimensional models. While the algorithm itself is not difficult to program, the substantial number of tuning parameters can be daunting to those unfamiliar with the theory behind the method. Until recently, practical access to HMC was limited to individuals with both the mathematical background to understand the algorithm and the programming skill to implement the simulation in a high-performance environment.

Modern Bayesian software such as Stan has made HMC accessible to practitioners who are comfortable with any one of a variety of well-known programming platforms (e.g. R, Python, Matlab). The Stan language is similar in style to WinBUGS, which is familiar to many Bayesian statisticians. The software translates Stan code to a lower-level language to maximize speed and efficiency. In addition, Stan automates the challenging process of tuning the many parameters in HMC. As a result, Stan has succeeded in making HMC accessible to many Bayesian practitioners around the world in both academia and industry.

While Stan and other high-performance software (e.g. PyMC, Edward) provide enormous practical value to analysts, the intuition of how HMC works can be lost in the process of fitting models. HMC can appear to be an opaque, "black-box" algorithm behind the sophisticated automation. This is an unfortunate consequence. While understanding HMC is not necessary to use production software, intuition behind the method can be helpful both in fine-tuning the simulation process and in instilling confidence in the results.

The purpose of this paper is to introduce HMC to analysts using R software only, an open-source statistical environment that is familiar to many. While many excellent introductions to HMC are available on a conceptual level, this paper will focus on learning HMC by doing. Familiarity with popular MCMC algorithms such as Metropolis-Hastings (MH) is helpful, but not required. A companion R package called hmclearn contains the R code for all of the functions used in this introduction is freely available to download.

# MCMC Basic Concepts

We consider $n$ observations from a simple random sample $\pmb{X} = (X_1, ..., X_n)$, where each element is independent and identically distributed (iid). From this sample, we want to fit the distribution of our k-dimensional parameter of interest $\Theta = (\theta_1, ..., \theta_k)$. The posterior distribution $p(\Theta | \pmb{X})$ can be written as a function of the Likelihood $p(\pmb{X} | \Theta)$ and prior $p(\Theta)$ using Bayes formula.

$$
\begin{aligned}
p(\Theta | \pmb{X}) &= \frac{p(\pmb{X}|\Theta)p(\Theta)}{\int p(\pmb{X}|\Theta)p(\Theta)d\Theta} \\
&\propto p(\pmb{X}|\Theta) p(\Theta)
\end{aligned}
$$

In many practical data analyses, the integral in the denominator cannot be evaluated directly. Since the denominator is constant with respect to $\Theta$, only the unnormalized posterior $p(\pmb{X} | \Theta)$ is available.

## Metropolis-Hastings

The first widely-used MCMC method capable of simulating directly from $p(\pmb{X} | \Theta)$ is called the Metropolis algorithm, originating in the 1950's from an application to statistical physics. Nearly two decades later, Hastings generalized the algorithm, which is now called Metropolis-Hastings (MH). We begin with a brief introduction to MH since HMC builds on many similar concepts.

The objective of MH is to simulate values of $\Theta$ that accurately

reflect the posterior density $p(\Theta | \pmb{X})$. For brevity, we will shorten our notation of the posterior as $p(\Theta)$. The Markov chain simulates values from this density in sequence from $t = 1, .., N$, provided some starting point $\Theta^{(0)}$ which is typically provided by the analyst or the computer program.
MH defines a transition probability that produces a Markov chain that is ergodic and satisfies detailed balance. Values of $\Theta^{(t)}$ in the chain are defined in part by a proposal density, which we define as $q(\Theta^{\text{PROP}} | \Theta^{t-1})$. Here, Θ^{PROP} is a proposal for the next value in the chain. This proposal density is conditioned on the previously stored value $\Theta^{(t-1)}$. A variety of proposal functions can be used, with random walk proposals being a common choice. We now outline the MH algorithm in full.

```
\begin{algorithm}
\caption{Metropolis-Hastings}\label{Metropolis-Hastings}
\begin{algorithmic}[1]
\Procedure{MH}{$\theta^{(0)}, \pi^{*}(\theta),
q(\theta^{(x)}|\theta^{(y)}), N$}
   \State Calculate $\pi^{*}(\theta^{(0)})$ \Comment{Initial value for
posterior}
   \For{$t = 1, ..., N$}\Comment{Repeat simulation $N$ times}
      \State $\theta^{\text{PROP}} \gets q(\theta^{\text{PROP}} |
\theta^{(t-1)})$ \Comment{Randomly sample proposal}
      \State $u \gets U(0, 1)$ \Comment{Randomly sample from a uniform
density, 0 to 1}
      \State $\alpha = \min\left(1,
\frac{\pi(\theta^{\text{PROP}})q(\theta^{\text{PROP}}|\theta^{(t-1)})}
{\pi(\theta^{(t-1)})q(\theta^{(t-1)}|\theta^{\text{PROP}})} \right)$
\Comment{Calculate acceptance proposal probability}
      \State If $\alpha < u$, then $\theta^{(t)} \gets
\theta^{\text{PROP}}$.  Otherwise, $\theta^{(t)} \gets \theta^{(t-1)}$
\Comment{Select proposal or previous value}
   \EndFor\label{markovendfor}
   \State \textbf{return} $\theta^{(1)} ... \theta^{(N)}$ \Comment{Return
simulated values of $\theta$ from the unnormalized posterior}
\EndProcedure
\end{algorithmic}
\end{algorithm}
```

Each proposal in MH is accepted at probability

$$
\alpha = \min\left(1,
\frac{p(\Theta^{\text{PROP}})q(\Theta^{\text{PROP}}|\Theta^{(t-1)})}
{p(\Theta^{(t-1)})q(\Theta^{(t-1)}|\Theta^{\text{PROP}})} \right)
$$

which simplifies when $q(\Theta^{PROP} | \Theta^{(t-1)})$ is symmetric (i.e. the Metropolis algorithm)

$$
\alpha = \min\left(1, \frac{p(\Theta^{\text{PROP}})}{p(\Theta^{(t-1)})}
\right).
$$

Recall that the denominator of the posterior is constant with respect to $\Theta$. As such, the ratio of posterior densities at two different points $\Theta^{PROP}$ and $\Theta^{(t-1)}$ can be formulated even when the denominator is unknown (i.e. the constants in the denominator cancel).

Intuition into why MH works can be obtained by examining the acceptance ratio $\alpha$ closely. Two different outcomes are possible depending on the value of the posterior at the proposed $\Theta^{PROP}$:

\begin{enumerate}
\item If $p(\Theta^{PROP}) \geq p(\Theta^{(t-1)})$, then the posterior has a higher density at the proposed value of $\Theta$ than at the previous point in the chain $t-1$. When this occurs, the proposal is always accepted (i.e. at probability 1).
\item If $p(\Theta^{PROP}) < p(\Theta^{(t-1)})$, then the posterior has a lower density at the proposed value of $\Theta$ than at the previous point in the chain. When this occurs, we accept the proposal at random based on the ratio $0 < \alpha < 1$. If the proposal is not accepted, then the proposal is discarded and the Markov chain remains in place $\Theta^{t} := \Theta^{(t-1)}$.
\end{enumerate}

As such, MH tends to sample more points in the region of higher posterior values. However, the tails of the posterior are also sampled based on acceptance ratio. Given enough samples, the MCMC chain samples $\Theta$ at the proportion of the true posterior density. The resulting simulated values can then be used for statistical inference. Much more can be said regarding MH. Interested readers can refer to the following references ... (list references here)

## Limitations of Metropolis-Hastings

The theoretical requirements for fitting models using MH are minimal, making MH an attractive choice for Bayesian inference even today. The limits of MH are primarily computational in nature. Since the proposals of $\Theta$ are randomly selected, many simulations are required to accurately describe the true posterior. Even efficient MH implementations may accept less than 25\% of the proposals (cite Gelman).

The limited efficiency of MH can be overcome by high computational power for certain applications. When the dimensionality of the data is small to moderate, a well-programmed MH algorithm can sample enough points from the posterior density in a reasonable period of time. The challenge of relying on MH occurs when dealing with high-dimensional data or complex statistical models. In these situations, MH is known to be inefficient (reference here) and can be impractical for such applications.

A popular, often efficient alternative to MH is Gibbs Sampling (footnote that Gibbs is a particular case of MH?). Gibbs is widely used in many Bayesian software platforms such as WinBUGS and JAGS. When the conditional posterior densities can be explicitly formulated, Gibbs remains a viable choice for the Bayesian practitioner. Such restrictions limit the application of Gibbs to particular combinations of models and priors. Gibbs therefore lacks the flexibility of MH, in addition to having certain other efficiency limitations of its own (cite Robert).

Given the modern computational demands of large dataset and complex models, a more efficient MCMC algorithm is desirable. Ideally, such an algorithm would retain the theoretical advantages and flexibility of MH, while providing a more computational efficient method of selecting proposals. Here we transition to the modern HMC algorithm, and why HMC has emerged as a standard inferential tool for many Bayesian practitioners.

# HMC Background

MH and HMC are equally flexible in their theoretical capabilities to fit a variety of model and prior specifications. The key advantage of HMC over MH is the use of additional information from the posterior to guide proposals. HMC uses the gradient of the log posterior to direct the Markov chain to the region the region of highest posterior density, where most of the samples should occur. In contrast, MH relies entirely on the acceptance ratio to guide the chain. As a result, a well-tuned HMC chain will accept proposals approximately 3 times the frequency of a similarly well-tuned MH algorithm (footnote?: based on MH theoretical optimal acceptance rate of 0.234 and HMC acceptance of 0.6-0.9).

It should be noted that HMC is a MCMC simulation method, not an optimization method. While the HMC algorithm guides proposals to regions of high density (sometimes called the typical set Betancourt), the tails of the density are properly sampled as well. Both MH and HMC produce ergodic Markov chains, but the mathematics of HMC is substantially more complex than MH. We provide a brief overview of the theoretical basis of HMC here, and refer to other sources for more detailed expositions (refer to Wiley paper, Betancourt).

## Hamiltonian Monte Carlo Concepts and Theory

As with MH, our objective in HMC is to simulate the posterior $p(\Theta | \pmb{X})$. The mathematical basis of HMC is the Hamiltonian function

$$
\begin{aligned}
H(\pmb{p}, \Theta) &= K(\pmb{p}, \Theta) + U(\Theta) \\
&= -\log p(\pmb{p}) - \log p(\Theta).
\end{aligned}

```
$$

HMC introduces a latent parameter $\pmb{p} = (p_1, ..., p_k)$ of the same
length $k$ as the parameter of interest $\Theta = (\theta_1, ...,
\theta_k )$.  The latent parameter $\pmb{p}$ is often called the
\textit{momentum} based on its original application to physical laws of
motion.  The incorporation of the momentum provides the geometrical
structure of the space through which the Markov chain travels.  The
purpose of the momentum is to ensure that the MCMC simulation is ergodic,
covering the entire space of $\Theta$.

The distribution of $\pmb{p}$ is most often specified to be multivariate
Normal with covariance $M$.  The structure of $M$ is often diagonal, but
does not have to be.

$$
\begin{aligned}
\pmb{p} &\sim N_k(0, M) \\
\log p(\pmb{p}) &\propto \frac{1}{2}\pmb{p}^T M^{-1}\pmb{p}
\end{aligned}
$$

The Hamiltonian function represents the joint distribution of the
multivariate Normal $\pmb{p}$ and the log posterior $p(\Theta |
\pmb{X})$, whose notation we simplify to $p(\Theta)$.

$$
\begin{aligned}
H(\pmb{p}, \Theta) &= -\frac{1}{2}\pmb{p}^T M^{-1} \pmb{p} - \log
p(\Theta)
\end{aligned}
$$
The trajectories over which HMC travels in time are defined by first-
order differential equations,

$$
\begin{aligned}
\frac{d\pmb{p}}{dt} &= -\frac{\partial H(\Theta, \pmb{p})}
{\partial\Theta}= -\frac{\partial U(\Theta)}{\partial\Theta} =
\nabla_\Theta \log p(\Theta) \\
\frac{d\Theta}{dt} &= \frac{\partial H(\Theta, \pmb{p})}{\partial\pmb{p}}
= \frac{\partial K(\pmb{p})}{\partial\pmb{p}} = M^{-1}\pmb{p}
\end{aligned}
$$
Here we see the explicit formulation of the gradient of the log posterior
$\nabla_\Theta \log p(\Theta)$.  The combination of these two equations
forms deterministic paths for the MCMC chain to travel through the joint
distribution $(\pmb{p}, \Theta)$.  Note that while the MCMC produces
simulations of both $\pmb{p}$ and $\Theta$, only the values of $\Theta$
are of interest for statistical inference.
```

A solution for these differential equations is necessary to produce a practical MCMC simulation. Since no exact solution exists, a discrete approximation is needed to form the algorithm. The most commonly used solution is called the leapfrog method. The leapfrog defines a discrete step size $\epsilon$ individually for $\pmb{p}$ and $\Theta$.

$$
\begin{aligned}
\pmb{p}(t + \epsilon/2) &= \pmb{p}(t) + (\epsilon/2)\nabla_\Theta\log\pi(\Theta(t)) \\
\Theta(t + \epsilon) &= \Theta(t) + \epsilon M^{-1}\pmb{p}(t + \epsilon/2) \\
\pmb{p}(t + \epsilon) &= \pmb{p}(t + \epsilon/2) + (\epsilon/2)\nabla_\Theta \log\pi(\Theta(t + \epsilon)).
\end{aligned}
$$

The full step $\epsilon$ in $\Theta$ is sandwiched by half-steps $\epsilon/2$ for $\pmb{p}$. The number of steps $\epsilon$ in an actual HMC algorithm is typically defined by a tuning parameter $L > 1$.   This leapfrog approximation, though slightly more complex than Euler approximations, provides a more accurate approximation of the solution over the many samples that HMC requires.

The paths defined by the Hamiltonian equations are deterministic once $\pmb{p}$ is defined. An exact solution would always be accepted in a MCMC algorithm. However, since the solution to the Hamiltonian equations is an approximation, a Metropolis-Hastings style acceptance ratio is used to formally determine whether a proposal is accepted or rejected.

With the overview of the major concepts in HMC complete, we turn to the formulation of the Hamiltonian Monte Carlo algorithm itself.

## Hamiltonian Monte Carlo Algorithm

The HMC algorithm is presented in full here.

```
\begin{algorithm}
\caption{Euclidean Hamiltonian Monte Carlo}\label{EHMC}
\begin{algorithmic}[1]
\Procedure{EHMC}{$\Theta^{(0)}, \log\pi(\Theta), M, N, \epsilon, L$}
   \State Calculate $\log\pi(\Theta^{(0)})$ \Comment{Initial value for log posterior}
   \For{$t = 1, ..., N$}\Comment{Repeat simulation $N$ times}
      \State $p^0 \gets N(0, M)$ \Comment{Randomly sample momentum from MVN}
      \State $\Theta^{(t)} \gets \Theta^{(t-1)}, \tilde{\Theta} \gets \Theta^{(t-1)}, \tilde{p} \gets p^{(0)}$ \Comment{Randomly sample from a uniform density, 0 to 1}
      \For{$i = 1, ..., L$}\Comment{Run Leapfrog $L$ times}
        \State $\tilde\Theta, \tilde{p} \gets \text{Leapfrog}
```

```
(\tilde\Theta, \tilde{p}, \epsilon)$
      \EndFor\label{leapfrogfor}
      \State $\alpha = \min{\left(1, \frac{\exp(\log\tilde\Theta) -
\frac{1}{2}\tilde{p}\cdot\tilde{p}}{\exp(\log\tilde\Theta^{(t-1)}) -
\frac{1}{2}p^0 \cdot p^0} \right)}$ \Comment{Calculate acceptance
proposal probability}
      \State With probability $\alpha$, $\Theta^{(t)}\gets\tilde\Theta$
and $p^{(t)} \gets -\tilde{p}$
   \EndFor\label{ehmcendfor}
   \State \textbf{return} $\Theta^{(1)} ... \Theta^{(N)}$ \Comment{Return
simulated values of $\Theta$ from the unnormalized posterior}
\Function{Leapfrog}{$\Theta, p, \epsilon$}
\State $\tilde{p} \gets p + (\epsilon/2)\nabla_\Theta\log\pi(\Theta)$
\State $\tilde\Theta \gets \Theta + \epsilon\tilde{p}$
\State $\tilde{p} \gets \tilde{p} +
(\epsilon/2)\nabla_\Theta\log\pi(\Theta)$
\State \textbf{return} $\tilde\Theta, \tilde{p}$
\EndFunction
\EndProcedure
\end{algorithmic}
\end{algorithm}

One may notice some similarities with MH:

\begin{enumerate}
\item An initial set of values $\Theta^{(0)}$ is required for both MH and
HMC.
\item The random walk Metropolis proposal is often Multivariate Normal.
HMC simulates from the momentum $\pmb{p}$ from a Multivariate Normal.
\item Proposals are accepted or rejected based on a ratio of log
posteriors in both MH and HMC.
\end{enumerate}

The key functional difference between the two algorithms is the leapfrog
update process.  HMC generates a proposal $(\pmb{p}^t, \Theta^t)$ based
on the randomly selected values of $\pmb{p}$ and the number of leapfrog
steps $L$.  In contrast, MH selects a proposal from one simple randomly
selected value from a proposal distribution.   MH is therefore
computationally simpler, but undirected.  HMC's proposal combines the
randomization provided by $\pmb{p}$ with trajectories defined the by the
gradient of the log posterior $\nabla_\Theta\log\pi(\Theta(t))$.

From an implementation standpoint, the additional programming required
for HMC is fairly minimal.  Manually determiming the gradient may be
tedious, but the derivation is typically not difficult.  Once the
gradient is provided and coded, programming the leapfrog loop itself is
straightforward.

The challenge of practical HMC implementation is related to
implementation of MH - tuning.  While tuning MH involves adjusting a
proposal density and little else, HMC requires the selection and
```

adjustment of the mass matrix $M$, number of leapfrog steps $L$, and step size $\epsilon$.

While $M$ can be fixed as an identity matrix, some modification of the diagonal, at a minimum, can provide notable improvements in computational efficiency. The selection of $L$ is also an essential step in HMC implementation. Selecting $L$ too low will produce a Markov chain that is inefficient, and may behave in pattern similar to random walk Metropolis, but with the penalty of additional computation. Setting $L$ too high can also be inefficient, where the chain may travel through the entire deterministic trajectory multiple times before a proposal is selected. Finally, $\epsilon$ can be set as a single constant for all $k$ parameters, but varying $\epsilon$ for each individual parameter may improve computational efficiency.

Developing an intuition and appreciation for how HMC works can be challenging without direct experience working with the algorithm. The next section provides R code and some simple examples for the interested reader to gain practical, hands-on experience with implementing and tuning HMC.

# HMC Implementation

We introduce several functions for implementing HMC in the R language. The function \textit{hmc} runs the main HMC algorithm. This function calls a separate function for the leapfrog algorithm and random value generation. Also, the user must specify functions to calculate both the log posterior as well as the gradient of the log posterior. These functions are used by both the main HMC function as well as the leapfrog function.

## hmc function

First we describe the main \textit{hmc} function itself.

```{r, echo=TRUE, eval=FALSE}
# Main hmc function
hmc <- function(N, theta.init, epsilon, L, logPOSTERIOR, glogPOSTERIOR,
y, X, Z=NULL,
                randlength=FALSE, Mdiag=NULL, verbose=FALSE, ...) {

  p <- length(theta.init) # number of parameters

  # mass matrix
  mu.p <- rep(0, p)

  # epsilon values
  eps_orig <- epsilon
  if (length(epsilon) == 1) {
    eps_orig <- rep(epsilon, p)
  }
  # epsilon matrix
  eps_vals <- matrix(rep(eps_orig, N), ncol=N, byrow=F)

  # number of steps
  nstep_vals <- rep(L, N)

  # randomize epsilon and L
  if (randlength) {
    randvals <- replicate(N, runif(p, -0.1*eps_orig, 0.1*eps_orig),
simplify=T)
    eps_vals <- eps_vals + randvals
    nstep_vals <- round(runif(N, 0.5*L, 2.0*L))
  }

  # default to unit diagonal matrix
  if (is.null(Mdiag)) {
    M_mx <- diag(p)
  }

  # invert covariance M for leapfrog
  Minv <- qr.solve(M_mx)

  # store theta and momentum (usually not of interest)
  theta <- list()
  theta[[1]] <- theta.init
  r <- list()
  r[[1]] <- NA
  accept <- 0
  for (jj in 2:N) {
    theta[[jj]] <- theta.new <- theta[[jj-1]]
    r0 <- MASS::mvrnorm(1, mu.p, M_mx)
    r.new <- r[[jj]] <- r0
    for (i in 1:L_vals) {
      lstp <- i == L_vals
      lf <- leapfrog(theta_lf = theta.new, r = r.new, epsilon = epsilon,
                     logPOSTERIOR = logPOSTERIOR,
                     glogPOSTERIOR = glogPOSTERIOR, y = y, X = X, Z = Z,
                     Minv=Minv, constrain=constrain, lastSTEP=lstp, ...)

      theta.new <- lf$theta.new
      r.new <- lf$r.new
    }

    if (verbose) print(jj)

    # standard metropolis-hastings update
    u <- runif(1)

    # use log transform for ratio due to low numbers
    num <- logDENS(theta.new, y=y, X=X, Z=Z, ...) - 0.5*(r.new %*% r.new)
```

~~~~
    den <- logDENS(theta[[jj-1]], y=y, X=X, Z=Z, ...) - 0.5*(r0 %*% r0)

    log.alpha <- pmin(0, num - den)

    if (log(u) < log.alpha) {
      theta[[jj]] <- theta.new
      r[[jj]] <- -r.new
      accept <- accept + 1
    } else {
      theta[[jj]] <- theta[[jj-1]]
      r[[jj]] <- r[[jj-1]]
    }

  }
  list(theta=theta,
       r=r,
       accept=accept,
       M=M_mx)
}


```

The arguments of \textit{hmc} are as follows:

\begin{itemize}
\item N:  Number of MCMC simulations
\item theta.init:  initial values $\Theta^{(0)}$ in vector of length $k$
\item epsilon:  step size $\epsilon$.  Can be a single value or vector
length $k$
\item L:  number of leapfrog steps per proposal
\item logPOSTERIOR:  name of function to return the log posterior
\item glogPOSTERIOR:  name of function to return the gradient of the log
posterior
\item y:  vector of the dependent variable
\item X:  design matrix of independent variables
\item randlength:  logical to choose whether to randomly vary number of
leapfrog steps $L$
\item Mdiag:  optional vector for the diagonal matrix $M$
\item verbose:  logical to print step in the MCMC chain
\item ...:  additional parameters for logPOSTERIOR and glogPOSTERIOR
\end{itemize}

Next, we review the functionality of the \textit{hmc} line-by-line.

HMC assigns a single latent parameter $p_i$ for each parameter $\theta_i
\; \forall i \in 1:k$.  Therefore, \textit{p} is assigned the number of
parameters in $\Theta$.

```{r, echo=TRUE, eval=FALSE}
  p <- length(theta.init)
~~~~

~~~
```

The distribution of the latent variables $\pmb{p}$ is multivariate Normal
with mean 0.  The mean of $\pmb{p}$ is assigned a vector 0 for each
$p_i$.

```{r, echo=TRUE, eval=FALSE}
  mu.p <- rep(0, p)
```

In this implementation, $\epsilon$ may be specified as a single number or
a vector.  If $\epsilon$ is a single number, it is converted to a vector
with a length of the number of parameters.


```{r, echo=TRUE, eval=FALSE}
  # epsilon values
  eps_orig <- epsilon
  if (length(epsilon) == 1) {
    eps_orig <- rep(epsilon, p)
  }

  # epsilon matrix
  eps_vals <- matrix(rep(eps_orig, N), ncol=N, byrow=F)
```

The number of steps $L$ can also be randomized.  In preparation for this
random assignment, the step sizes for each iteration $1...N$ are stored
in a vector \textit{L\_vals}.

```{r, echo=TRUE, eval=FALSE}
  # number of steps
  L_vals <- rep(L, N)

```

The \textit{randlength} is an optional parameter to randomize $\epsilon$
and $L$.  If this is set to \textit{TRUE}, then a random uniform
distribution is used to add some randomness to $\epsilon$.  The number of
steps $L$ is similarly randomized, but restricted to integer values.

```{r, echo=TRUE, eval=FALSE}
  # randomize epsilon and L
  if (randlength) {
    randvals <- replicate(N, runif(p, -0.1*eps_orig, 0.1*eps_orig),
simplify=T)
    eps_vals <- eps_vals + randvals
    L_vals <- round(runif(N, 0.5*L, 2.0*L))
  }
```
~~~

If the Mass matrix is not provided, the identity matrix is used as a default.

~~~
```{r, echo=TRUE, eval=FALSE}
  # default to unit diagonal matrix
  if (is.null(Mdiag)) {
    M_mx <- diag(p)
  }
```
~~~

In preparation for the leapfrog algorithm, the Mass matrix is inverted.

~~~
```{r, echo=TRUE, eval=FALSE}
  # invert covariance M for leapfrog
  Minv <- qr.solve(M_mx)
```
~~~

Recall that our objective is to simulate values of $\Theta$ from the posterior. In this function, the simulated values will be stored in a list \textit{theta}. The first element of the list \textit{theta[[1]]} is assigned to the initial value provided in the function.

~~~
```{r, echo=TRUE, eval=FALSE}
  theta <- list()
  theta[[1]] <- theta.init
```
~~~

A second list is created to store simulated values of the momentum $\pmb{p}$. These simulated values are typically not of interest in analysis, but can be useful in debugging. The first value of $\pmb{p}$ is assigned \textit{NA} for the initial value of $\Theta$ only.

~~~
```{r, echo=TRUE, eval=FALSE}
  r <- list()
  r[[1]] <- NA
```
~~~

The acceptance rate of HMC (and MH) is an important measure in assessing the efficiency of the MCMC simulation. An internal variable \textit{accept} is initialized to zero and is used to count the number of accepted proposals.

~~~
```{r, echo=TRUE, eval=FALSE}
  accept <- 0
```
~~~

With the initialization complete, we begin the loop to perform the HMC simulations. The loop starts at 2 since 1 is reserved for the initial value $\Theta^{(0)}$.

~~~
```{r, echo=TRUE, eval=FALSE}
  for (jj in 2:N) {
```

The default proposed value of $\Theta^{(t)}$ is the previous value in the
chain, $\Theta^{(t-1)}$.  This value may be replaced with the proposal
depending on the acceptance ratio calculation later in the loop.

```{r, echo=TRUE, eval=FALSE}
    theta[[jj]] <- theta.new <- theta[[jj-1]]
```

Each proposal requires a simulated value of $\pmb{p}$.  The standard
\textit{mapply} function passes the mean vector \textit{mu.p} and
standard deviation vector \textit{Mdiag} to the \textit{rnorm} function.
The additional argument \textit{n = 1} is passed to \textit{rnorm} to
simulate one value of $p_i$ for each combination of mean and standard
deviation in the corresponding vectors.

The \textit{mvrnorm} function in the \textit{MASS} package is used to
simulate a multivariate Normal with mean \textit{mu.p} and covariance
matrix \textit{M\_mx}

```{r, echo=TRUE, eval=FALSE}
  r0 <- MASS::mvrnorm(1, mu.p, M_mx)
```

The initial sampled value of $p_t$ is stored in the momentum list
\textit{r[[m]]}.  This value will be replaced if the proposal is accepted
later in the loop.

```{r, echo=TRUE, eval=FALSE}
    r.new <- r[[jj]] <- r0
```

With the previous value $\Theta^{(t-1)}$ and simulated momentum, the
leapfrog can be evaluted $L$ consecutive times.  The proposal will be the
joint value of $(\pmb{\tilde{p}}^{t}, \tilde\Theta^{(t)})$ after
progressing through the $L$ leapfrog steps.

The simulated values for each leapfrog step are stored in
\textit{theta.new} and \textit{r.new}.  After $L$ steps, these variables
store the proposal.

```{r, echo=TRUE, eval=FALSE}
    for (i in 1:L) {
      lstp <- i == Nstep
      lf <- leapfrog(theta_lf = theta.new, r = r.new, epsilon = epsilon,
                     logPOSTERIOR = logPOSTERIOR,
                     glogPOSTERIOR = glogPOSTERIOR, y = y, X = X, Z = Z,
                     Minv=Minv, constrain=constrain, lastSTEP=lstp, ...)
~~~

~~~
      theta.new <- lf$theta.new
      r.new <- lf$r.new
    }
```
~~~

Next is an optional counter to print the count in the \textit{for} loop.

~~~
```{r, echo=TRUE, eval=FALSE}
    if (verbose) print(jj)
```
~~~

The proposal is then evaluated in a Metropolis-Hastings style acceptance step. First, a standard uniform random value is simulated and stored in \textit{u}. The proposal will be accepted if the acceptance ratio is less than this value.

~~~
```{r, echo=TRUE, eval=FALSE}
    u <- runif(1)
```
~~~

A log transformation is used to calculate the acceptance ratio. This transformation is used for numerical stability, since the simulated values of the log posterior can be very high. The log ratio is set to a maximum of 0 since $\log(1) = 0$ is the maximum value in the range $u \in [0, 1]$.

~~~
```{r, echo=TRUE, eval=FALSE}
    num <- logPOSTERIOR(theta.new, y=y, X=X, Z=Z, ...) - 0.5*(r.new %*%
r.new)
    den <- logPOSTERIOR(theta[[jj-1]], y=y, X=X, Z=Z, ...) - 0.5*(r0 %*%
r0)

    log.alpha <- pmin(0, num - den)
```
~~~

Finally, a check is performed on the acceptance ratio. If $\alpha < u$ (or $\log\alpha < \log u$), then the proposal is accepted. Otherwise, the previous value in the MCMC chain is retained.

If the proposal is accepted, the negative value of momentum is stored in \textit{r[[m]]}. This is a technicality to assure that the MCMC chain follows detailed balance (cite here), but this value is not used for inference. Also, the \textit{accept} variable is incremented by one when the proposal is accepted.

~~~
```{r, echo=TRUE, eval=FALSE}
    if (log(u) < log.alpha) {
      theta[[jj]] <- theta.new
      r[[jj]] <- -r.new
      accept <- accept + 1
~~~

~~~~
    } else {
      theta[[jj]] <- theta[[jj-1]]
      r[[jj]] <- r[[jj-1]]
    }
```

After \textit{N} simulations, a list of results is exported from
\textit{hmc}:

\begin{enumerate}
\item theta:  list of simulated values of $\Theta$
\item r:  list of simulated values of the momentum $\pmb{p}$
\item accept:  the number of accepted proposals in \textit{N} simulations
\item M\_mx:  Mass matrix
\end{enumerate}

```{r, echo=TRUE, eval=FALSE}
  list(theta=theta,
       r=r,
       accept=accept,
       M=M_mx)
```

## Leapfrog function

Next, we review the leapfrog function in more detail.

```{r, echo=TRUE, eval=FALSE}
leapfrog <- function(theta_lf, r, epsilon, logPOSTERIOR, glogPOSTERIOR,
y, X, Z,
                     Minv, constrain, lastSTEP=FALSE, ...) {

  # gradient of log posterior for old theta
  g.ld <- glogPOSTERIOR(theta_lf, y=y, X=X, Z=Z, ...)

  # first momentum update
  r.new <- r + epsilon/2*g.ld

  # theta update
  # note diagonal matrix update
  # theta.new <- theta_lf + as.numeric(epsilon*r.new/ diag(M_mx))
  theta.new <- theta_lf + as.numeric(epsilon* Minv %*% r.new)

  # check positive
  switch_sign <- constrain & theta.new < 0
  r.new[switch_sign] <- -r.new[switch_sign]
  theta.new[switch_sign] <- -theta.new[switch_sign]

  # gradient of log posterior for new theta
  g.ld.new <- glogPOSTERIOR(theta.new, y=y, X=X, Z=Z, ...)
~~~~

~~~~
  # if not on last step, second momentum update
  if (!lastSTEP) {
    r.new <- r.new + epsilon/2*g.ld.new
  }
  list(theta.new=theta.new,
       r.new=as.numeric(r.new))
}

```

The arguments of \textit{leapfrog} are as follows:

\begin{itemize}
\item theta\_lf:  Starting position of $\Theta^{(t)}$
\item r:  Starting momentum $p$
\item epsilon:  step size $\epsilon$.  Can be a single value or vector
length $k$
\item logPOSTERIOR:  name of function to return the log posterior
\item glogPOSTERIOR:  name of function to return the gradient of the log
posterior
\item y:  vector of the dependent variable
\item X:  design matrix of independent variables
\item Z:  optional design matrix for random effects
\item constrain:  logical indicator of whether the support of $\Theta$ is
positive only (TRUE) or all real numbers (FALSE)
\item lastSTEP: logical indicator of whether this is the last step of the
leapfrog loop.  If it is, then the last half step evaluation for the
momentum is not evaluated.
\item ...:  additional parameters for logPOSTERIOR and glogPOSTERIOR
\end{itemize}

Next, we review the functionality of the \textit{leapfrog} line-by-line.

First, the gradient of the log posterior is evaluated at the initial
value provided for $\Theta^{(t)}$.  If \textit{Z} is not provided, a NULL
value will be passed to this parameter.  Hyperparameters are passed to
the \textit{glogPOSTERIOR} function via \textit{...}.

```{r, echo=TRUE, eval=FALSE}
  # gradient of log posterior for old theta
  g.ld <- glogPOSTERIOR(theta_lf, y=y, X=X, Z=Z, ...)
```

Once the gradient is calculated, the first update of the momentum is
performed.  The step size for this first update is half of the step size
$\epsilon$.

```{r, echo=TRUE, eval=FALSE}
  # first momentum update
  r.new <- r + epsilon/2*g.ld
```
~~~~

After the momentum half-step update, $\Theta$ is updated by the full step size $\epsilon$.

```{r, echo=TRUE, eval=FALSE}
  # theta update
  theta.new <- theta_lf + as.numeric(epsilon* Minv %*% r.new)
```

The next few steps are optionally performed to change the sign of the momentum and $\Theta$ if the support of $\Theta$ is strictly positive (cite Neil). These steps are only relevant if \textit{constrain} is set to TRUE. Otherwise, the momentum and $\Theta$ are unchanged.

```{r, echo=TRUE, eval=FALSE}
  # check positive
  switch_sign <- constrain & theta.new < 0
  r.new[switch_sign] <- -r.new[switch_sign]
  theta.new[switch_sign] <- -theta.new[switch_sign]
```

Once $\Theta$ is updated, the gradient of the log posterior is re-calculated at the new set of values.

```{r, echo=TRUE, eval=FALSE}
  # gradient of log posterior for new theta
  g.ld.new <- glogPOSTERIOR(theta.new, y=y, X=X, Z=Z, ...)
```

Finally, unless this the final leapfrog step, the momentum is updated by its second half-step.

```{r, echo=TRUE, eval=FALSE}
  # if not on last step, second momentum update
  if (!lastSTEP) {
    r.new <- r.new + epsilon/2*g.ld.new
  }
```

The \textit{leapfrog} function then returns the new values of $\Theta$ and momentum in a list.

```{r, echo=TRUE, eval=FALSE}
  list(theta.new=theta.new,
       r.new=as.numeric(r.new))
```

## Simple example

To illustrate the functionality of the HMC software, we begin with a simple example fitting a Gamma distribution with two parameters $\alpha$

and β. The pdf is specified

$$
f(x) = \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1}e^{-
x/\beta}I_x(0, \infty)
$$

with likelihood and log likelihood

$$
\begin{aligned}
L(\alpha, \beta; x) &= \prod_{i=1}^n \frac{1}{\beta^\alpha
\Gamma(\alpha)} x_i^{\alpha-1}e^{-x_i/\beta} \\
l(\alpha, \beta;x) &= \sum_{i=1}^n -\alpha\log\beta -\log\Gamma(\alpha) +
(\alpha-1)\log x_i - x_i/\beta \\
&= -n\alpha\log\beta -n\log\Gamma(\alpha) + (\alpha-1)\sum_{i=1}^n\log
x_i - \frac{1}{\beta}\sum_{i=1}^n x_i.
\end{aligned}
$$

Now that the log likelihood has been specified, we can specify our
priors. Both parameters α and β are strictly positive. As
such, we can specify half-Normal priors for each of the parameters, with
hyperpriors η_1 and η_2.

$$
\begin{aligned}
p(\alpha|\eta_1) &= \frac{2\eta_1}{\pi}\exp\left(-\frac{\alpha^2\eta_1^2}
{\pi} \right) \\
p(\beta|\eta_2) &= \frac{2\eta_2}{\pi}\exp\left(-\frac{\beta^2\eta_2^2}
{\pi} \right)
\end{aligned}
$$

The log posterior is the sum of the log likelihood and the log prior.

$$
\begin{aligned}
\log p(\alpha, \beta|x) &\propto l(\alpha, \beta; x) + \log
p(\alpha,\beta) \\
&\propto -n\alpha\log\beta -n\log\Gamma(\alpha) + (\alpha-
1)\sum_{i=1}^n\log x_i - \frac{1}{\beta}\sum_{i=1}^n x_i -
\frac{\alpha^2\eta_1^2}{\pi} - \frac{\beta^2\eta_2^2}{\pi}
\end{aligned}
$$

Note that we disregard any terms that are not dependent on our parameters
of interest α and β. These terms are absorbed into the
normalizing constant and do not impact the gradient calculation.

The R functions for the log likelihood and log posterior are provided

~~~~
here.  For our application, our parameter of interest is defined $\Theta
:= (\alpha, \beta)$.

```{r, echo=TRUE, eval=TRUE}
# log likelihood of gamma
llgamma <- function(theta, X, y=NULL, Z=NULL) {
  alpha <- theta[1]
  beta <- theta[2]
  n <- length(X)
  -n*alpha*log(beta) - n*log(gamma(alpha)) + (alpha-1)*sum(log(X)) -
sum(X)/beta
}

# log posterior of gamma
gamma.lposterior <- function(theta, X, eta1, eta2, y=NULL, Z=NULL) {
  alpha <- theta[1]
  beta <- theta[2]
  llgamma(theta, X) - beta^2 * eta1^2/pi - alpha^2 * eta2^2/pi
}
```


Since our model has two parameters, the partial derivatives must be
calculated with respect to $\alpha$ and $\beta$.

$$
\begin{aligned}
\nabla_\alpha \log p(\alpha, \beta|x) &= -n\log\beta -n\psi(\alpha) +
\sum_{i=1}^n\log x_i -2\alpha\eta_1^2/\pi \\
\nabla_\beta \log p(\alpha, \beta|x) &= -\frac{n\alpha}{\beta} + \frac{1}
{\beta^2}\sum_{i=1}^n x_i - 2\beta\eta_2^2/\pi
\end{aligned}
$$

The R function for the gradient of the log posterior is provided here.

```{r, echo=TRUE, eval=TRUE}
# derivative of posterior
g.gamma.lposterior <- function(theta, X, eta1, eta2) {
  alpha <- theta[1]
  beta <- theta[2]
  n <- length(X)
  dalpha <- -n*log(beta) -n*digamma(alpha) + sum(log(X)) -
2*alpha*eta1^2/pi
  dbeta <- -n*alpha/beta + sum(X)/beta/beta - 2*beta*eta2^2/pi
  c(dalpha, dbeta)
}
```

Now that the log posterior and gradient have been derived and coded in R,
we can run the \textit{hmc} function to fit the model.
~~~~

~~~
First, we simulate data based on a gamma distribution.  Here, we simulate
1000 data points with $\alpha = 2$ and $\beta = 3$.

```{r, echo=TRUE, eval=TRUE}
# simulate data
set.seed(312)
X <- rgamma(1000, 2, 1/3)
```

Next, we run our $\textit{hmc}$ function on our simulated data.  For this
example, we set the initial values $\Theta^{(0)}$ sufficiently away from
zero with pre-selected parameters $\epsilon$ and $L$.  This is to prevent
the MCMC chain from moving to values less than zero.

In production software, a transformation is typically performed to the
parameters to allow values from the entire real number line (e.g. log
transform).  Alternatively, a \textit{constrain} parameter can be set to
designate $\alpha$ and $\beta$ as strictly positive.

```{r, echo=TRUE, eval=TRUE}
N <- 10000
set.seed(143)
ex1 <- hmc(N, theta.init = c(4, 4), epsilon = 2e-2, L = 22,
                   logPOSTERIOR = gamma.lposterior,
                   glogPOSTERIOR = g.gamma.lposterior, X=X,
                   varnames = c("alpha", "beta"),
                   eta1 = 1e-4, eta2 = 1e-4)
```

The \textit{hmclearn} package has a diagnostic plot function to show
trace plots and histograms from the simulation.  An optional parameter
\textit{actual.mu} can be used to input the true values of the parameters
from simulated data.  A default \textit{burnin} period of 100 samples is
also selected for these plots.

Without performing rigorous diagnostics, the trace plots appear
stationary with the given burnin period.  The histograms of the
simulations also appear well within the range of highest posterior
density.

```{r, echo=TRUE}
plot(ex1, actual.mu = c(2, 3))

```

A \textit{summary} confirms that the actual parameter values of $\Theta$
fall well within the simulated posterior values.

```{r, echo=TRUE, eval=TRUE}
summary(ex1)
~~~

~~~~
```

In the next section, we will discuss some of the main practical
considerations in fitting models using HMC, focusing on parameterization
and tuning.

# HMC Practical Considerations

Parameters for some statistical models are bounded.  When parameters are
bounded, the MCMC chain is restricted on simulated values.  Since the HMC
algorithm approximates the trajectories through the joint space of the
momentum and $\Theta$, it is possible that the chain may move to a
location outside of the support of $\Theta$.

## Constrained Parameters

Parameters for many statistical models are limited to strictly positive
numbers.  This was true for the first example using HMC to fit $\alpha$
and $\beta$ from a Gamma distribution.

We continue our previous example by implementing log transformations for
each of these parameters.

$$
\begin{aligned}
a &= \log\alpha \\
p_a(a) &= p_\alpha(g^{-1}(a))\left|\frac{d\alpha}{da}\right| \\
&= p_\alpha(e^a)e^a \\
\log p_a(a) &= \log p_\alpha(e^a) + a \\
&= \log p_\alpha(\alpha) + \log\alpha
\end{aligned}
$$

The Jacobian term $\log\alpha$ must be included with the prior when
employing the log transformation of the original parameter.  A similar
transformation is necessary for $\beta$.

$$
\begin{aligned}
b &= \log\beta \\
\log p_b(b) &= \log p_\beta(\beta) + \log\beta
\end{aligned}
$$

The log posterior can be re-written with the new parameterization.


$$
\begin{aligned}
\log p(a, b|x) &\propto l(\alpha, \beta; x) + \log p(\alpha,\beta) \\
&\propto l(\alpha, \beta; x) + \log p(\alpha) + \log\alpha + \log
~~~~

~~~
p(\beta) + \log\beta \\
&\propto  -n\alpha\log\beta -n\log\Gamma(\alpha) + (\alpha-
1)\sum_{i=1}^n\log x_i - \frac{1}{\beta}\sum_{i=1}^n x_i -
\frac{\alpha^2\eta_1^2}{\pi} - \frac{\beta^2\eta_2^2}{\pi} + \log\alpha +
\log\beta
\end{aligned}
$$

The gradient includes the partial derivatives of the Jacobian terms as
well.

$$
\begin{aligned}
\nabla_\alpha \log p(a, b|x) &= -n\log\beta -n\psi(\alpha) +
\sum_{i=1}^n\log x_i -2\alpha\eta_1^2/\pi + \frac{1}{\alpha}\\
\nabla_\beta \log p(a, b|x) &= -\frac{n\alpha}{\beta} + \frac{1}
{\beta^2}\sum_{i=1}^n x_i - 2\beta\eta_2^2/\pi + \frac{1}{\beta}
\end{aligned}
$$

Revised R functions are required for the log posterior and gradient.

```{r, echo=TRUE, eval=TRUE}

# log likelihood of gamma with log-transformed parameters
llgamma2 <- function(theta, X, y=NULL, Z=NULL) {
  a <- theta[1]
  b <- theta[2]
  alpha <- exp(a)
  beta <- exp(b)
  n <- length(X)
  -n*alpha*log(beta) - n*log(gamma(alpha)) + (alpha-1)*sum(log(X)) -
sum(X)/beta
}

# log posterior of gamma with log-transformed parameters
gamma.lposterior2 <- function(theta, X, eta1, eta2, y=NULL, Z=NULL) {
  a <- theta[1]
  b <- theta[2]
  alpha <- exp(a)
  beta <- exp(b)
  llgamma2(theta, X) - beta^2 * eta1^2/pi - alpha^2 * eta2^2/pi +
log(alpha) + log(beta)
}

# derivative of posterior with log-transformed parameters
g.gamma.lposterior2 <- function(theta, X, eta1, eta2) {
  a <- theta[1]
  b <- theta[2]
  alpha <- exp(a)
  beta <- exp(b)
~~~

~~~
  n <- length(X)
  dalpha <- -n*log(beta) -n*digamma(alpha) + sum(log(X)) -
2*alpha*eta1^2/pi + 1/alpha
  dbeta <- -n*alpha/beta + sum(X)/beta/beta - 2*beta*eta2^2/pi + 1/beta
  c(dalpha, dbeta)
}
```

Now, we can safely re-fit the model with parameters that span the full
real number line.  Note that we are re-defining $\tilde\Theta := (a, b) =
(\log\alpha, \log\beta)$ with initial values at zero.

```{r, echo=TRUE, eval=TRUE}
N <- 10000
set.seed(143)
ex1b <- hmc(N, theta.init = c(0, 0), epsilon = 1e-3, L = 30,
                   logPOSTERIOR = gamma.lposterior2,
                   glogPOSTERIOR = g.gamma.lposterior2, X=X,
                   varnames = c("a", "b"),
                   eta1 = 1e-4, eta2 = 1e-4)
```

HMC accurately fits the model using the log-transformed parameters, as we
can see through the diagnostic plots and summary results.

```{r, echo=TRUE}
plot(ex1b, actual.mu = c(log(2), log(3)))

```

```{r, echo=TRUE}
summary(ex1b)
```

The exponential of the results confirms that the HMC converges to the
correct parameters.

```{r, echo=TRUE}
exp(summary(ex1b))
```

Parameter transformation such as the $log$ transform shown here is the
most common method of ensuring that the MCMC simulation is remains in the
support of $\Theta$.  An alternate approach to constrained variables was
proposed by Neil (cite).  This approach modifies the \textit{leapfrog}
function to ensure the proposed $\Theta$ remains within the appropriate
support.  A simplifed version of this method is provided in the
\textit{hmc} function for strictly positive parameters (i.e. with support
$(0, \infty)$).

\begin{itemize}
~~~

```
\item[] Initial momentum half-step:  $\pmb{p}'(t + \epsilon/2) = \pmb{p}
(t) + (\epsilon/2)\nabla_\Theta\log\pi(\Theta(t))$
\item[] Initial full-step of $\Theta$:  $\Theta'(t + \epsilon) =
\Theta(t) + \epsilon M^{-1}\pmb{p}'(t + \epsilon/2)$
\item[] Check lower constraint of $0$:
\begin{itemize}
\item[] If $\Theta'(t + \epsilon) < 0$ then $p(t + \epsilon/2) = -p'(t +
\epsilon/2)$ and  $\Theta(t + \epsilon) = -\Theta'(t + \epsilon)$.
Otherwise, keep the proposed momentum and $\Theta$ as simulated
\end{itemize}
\end{itemize}

## Tuning

Metropolis-Hastings and HMC each utilize parameters that must be tuned to
efficiently simulate from the posterior distribution.  This is a matter
of more practical importance than theoretical necessity.  In the limit,
both MCMC algorithms will converge to the true posterior given infinite
samples.  However, the converge rate of these algorithms varies
substantially depending on how well the tuning parameters fit the
application.

Tuning for random-walk Metropolis (RWM) typically involves adjusting the
variance parameterization of the proposal density.  Standardizing the
design matrix to a common mean and variance can significantly simplify
the tuning exercise.  For example, a multivariate Normal proposal may
simply involve adjusting a scaler for the identity covariance matrix.
The analyst typically examines preliminary trace plots along with
acceptance rate calculations to assess tuning effectiveness.

HMC can be significantly more complex than RWM due to the number and
variety of tuning parameters in the HMC algorithm.  Like RWM with a
multivariate Normal proposal, HMC uses a multviariate Normal
parameterization for the latent momentum variable $p$.  While HMC may use
an identity covariance matrix for $M$, tuning the diagonal and even off-
diagonal parameters can improve the efficiency of sampling.

In addition to the covariance matrix $M$, HMC requires parameter
selections for the step size $\epsilon$ and number of leapfrog steps $L$.
Setting $\epsilon$ and $L$ to low values, relative to the particular
application, typically result in a high acceptance rate (e.g. greater
than 95\%).  On the surface, accepting the vast majority of proposals may
appear to be desirable.  However, the combination of a small step size
with few leapfrog steps can create a MCMC chain that traverses the
support of $\Theta$ very slow.  The evidence of inappropriately small
values for these tuning parameters is a trace plot that exhibits a high
degree of autocorrelation.

Generally, the step size should be set as high as possible to balance the
acceptance rate and minimize autocorrelation.  An acceptance rate of
approximately 60 - 90\% often produces MCMC chains that converge
```

sufficiently quickly. While $\epsilon$ and $L$ may be tuning jointly, the step size is often selected first, followed by fine-tuning with the number of steps per leapfrog $L$.

A general approach to tuning HMC is provided here:

\begin{enumerate}
\item Set the stepsize to an initial value such as $\epsilon = 1e-2$ with $L = 10$ and $M$ unit diagonal
\item Run a preliminary HMC chain and compute the acceptance rate. Adjust $\epsilon$ until the acceptance is between 0.6 and 0.9.
\item Check the autocorrelation for each parameter either visually or via direct calculation. For parameters with high autocorrelation, reduce the relevant value in the diagonal of $M$. This adjustment may decrease the acceptance rate.
\item If necessary, increase $L$ to further reduce autocorrelation in the simulation
\end{enumerate}

Additional adjustments may be made to the tuning parameters beyond these steps. The value of $\epsilon$ may be constant for each parameter in $(\theta_1, ..., \theta_k)$ in $\Theta$, but does not need to be. The analyst may instead provide a vector $\epsilon:= \epsilon_1, ..., \epsilon_k$ with a different value for each $\epsilon_i \forall\; i \in 1...k$. In the \textit{hmclearn} package, the \textit{hmc} function accepts single values and vectors for the parameter \textit{epsilon}.

The parameter for the number of steps $L$ must be a natural number. However, the number of steps may be randomized in each simulation $1...N$ to ensure against a periodic Markov chain (which would violate the regularity conditions for ergodicity). Further, some randomness may be applied to $\epsilon$ in each simulation step $1...N$ for additioanl assurance of an aperiodic chain. Randomness can be automatically applied for $\epsilon$ and $L$ with logical parameter \textit{randlength} in the \textit{hmc} function.

Finally, we note that the popular No-U-Turn Sampler (NUTS) algorithm (cite paper) is a popular automated option for selecting $L$. We avoid using NUTS in this paper as the emphasis here is more pedagogical. The intent is to provide interested readers with tools to experiment with the core HMC algorithm to gain intuition on the particulars of the algorithm.

## QR Decomposition

The efficiency of sampling in the standard HMC algorithm can be improved significantly for multivariate models when the parameters $(\theta_1, ..., \theta_k) \in \Theta$ have an orthogonal basis. One common method of ensuring an orthogonal basis involves applying QR decomposition to the design matrix prior to applying HMC.

For example, Stan scales QR decomposition with a factor of $\sqrt{n-1}$

```
to ensure unit variance for continuous parameters (cite stan manual).
Here $X$ represents the design matrix for the model

$$
\begin{aligned}
X &= QR = Q^*R^* \\
Q^* &= \sqrt{n-1} Q \\
R^* &= \frac{1}{\sqrt{n-1}}R \\
X\Theta &= Q^*R^*\Theta \\
&= Q^* \widetilde{\Theta}
\end{aligned}
$$

The scaled $Q$ matrix is used in the HMC simulation in place of the raw
design matrix $X$.  The simulated parameters will then be from
$\widetilde{\Theta}$.  To transform back to the original scale, we
multiply by the inverse of $R^*$

$$
\begin{aligned}
R^*\Theta &= \widetilde{\Theta}\\
\Theta &= R^{*-1}\widetilde{\Theta}
\end{aligned}
$$

In R software, the base functions \textit{qr} can be used to perform QR
decomposition.  The $Q$ and $R$ matrices can be obtained from the R
methods \textit{qr.Q} and \textit{qr.R}, respectively, applied to the
resulting object.

# HMC in Statistical Models

With a background of the HMC algorithm and its application to simple
examples complete, we turn to more practical, real-world examples.
Generalized Linear Models (GLM) are a broad class of statistical models
that use a linear function to relate the mean response to a set of
independent variables.  Nonlinear functions of the mean can be
accommodated through a specified link function.  While there are the
number and types of models to which HMC can be applied are limitless, we
will focus on GLM's in this introductory article.

The major steps required to fit a statistical model are summarized below.
We will elaborate on what is required for each step in this process.

\begin{enumerate}
\item Specify model
\item Derive log posterior
\item Re-parameterize parameters as needed (i.e. log transforms)
\item Derive gradient of the log posterior
\item Code functions for log posterior and gradient
\item Tune HMC parameters with trial runs
```

```
\item Run HMC
\end{enumerate}

## Step 1:  Specify model

The general form of a GLM can be specified (cite Agresti GLM book)

$$
g[E(y)] = \pmb{X}\bm{\beta}
$$

Here, we consider $\bm{\beta}$ our parameter of interest, such that
$\Theta := \bm{\beta}$.  The linear predictor of the GLM is the product
of the design matrix $(\pmb{x}_1, ..., \pmb{x}_k) \in \pmb{X}$ for $i =
1, .., n$ observations, and the parameter vector $(\beta_1, ..., \beta_k)
\in \bm{\beta}$.  A link function $g(\cdot)$ relates the linear predictor
to the mean response.  For a linear regression model, $g(\cdot)$ is the
identity function, such that $E(y) = \pmb{X}\bm{\beta}$.

## Step 2:  Derive log posterior

The log posterior is comprised of the sum of the log likelihood of the
model and the log prior.    In general, the likelihood can be written
based on the inverse of the selected link function.
$$
f(y; \bm{\beta}) = g^{-1}(\pmb{X}\bm{\beta}) = p(y | \pmb{X}\bm{\beta})
$$

The likelihood functions for standard GLM's are readily available from
statistical texts on the subject.

Once the likelihood is defined, the prior must be selected for the
parameter of interest.  Unlike more restrictive algorithms such as Gibbs,
the analyst has a substantial degree of flexibility in prior selection
for HMC.  Depending on the application, the form of the prior may be
defined as restricted within a certain range when known in advance.

Alternatively, the prior is defined as vague over the support of
$\bm{\beta}$.  For example, when $\bm{\beta}$ may be any real number, a
vague prior may be a multivariate Normal with high variance.  The
resulting prior density would be flat over all practical values of the
parameter of interest.

Once the log likelihood is defined and the prior is selected, the log
posterior may be written

$$
\log p(\bm\beta | y, \pmb{X}) = \log p(y | \pmb{X}\bm{\beta}) + \log
p(\bm{\beta})
$$
```

## Step 3: Re-parameterize parameters as needed (i.e. log transforms)

This step should always be considered, but may not be necessary depending on the particular model. Typically, any standard deviation or variance parameters would be restricted to positive real numbers. These would need to either be re-parameterized or the constrained HMC leapfrog would need to be used.

A logistic regression model, for example, would only have the parameter vector for $\bm{\beta}$, for which the support can be the entirely real number line. A linear regression model has an error term $\sigma^2$ which is strictly positive. Fitting an HMC without considering the support of $\sigma^2$ could result in the MCMC chain selecting a negative sample. In such a case, the samples would no longer be valid for the parameter of interest and the algorithm would likely produce an error.

When re-parameterization is necessary, it is important to include the Jacobian adjustment for new parameter. Detailed explanation of a change of variables can be found in standard statistical texts.

## Step 4: Derive gradient of the log posterior

Since the gradient is a linear operator, the gradient of the log posterior is equivalent to the sum of the gradient of the log likelihood and the gradient of the log prior. One may derive the gradient of the log likelihood and log prior each individually, in preparation for coding functions for these elements individually.

Note that if testing multiple prior distributions is desired, the gradient must be derived for each prior distribution to be tested.

## Step 5: Code functions for the log posterior and gradient

Once the log posterior and gradient are formed, they must be coded into separate R functions. This process often requires some iteration and debugging, particularly for complex likelihood functions.

While the forms of the log likelihood and log prior can often be found from standard sources, the gradient may not be as readily available. As such, the analyst should test the gradient function at several values to ensure that it is returning the correct result.

One approach to testing the accuracy of a custom gradient function is to use a package that uses finite differencing or some variant to approximate the gradient given a general function. The R package \textit{pracma} contains functions that can be useful in this regard.

While finite differencing is appropriate for checking the accuracy of coding the gradient, such approximations are typically not sufficiently accurate for use in an HMC simulation. Further, gradient approximation is typically slower than a direct gradient function. Since the gradient must be calculated for each iteration of the leapfrog algorithm, a slower approximation function can substantially increase the computation time for HMC.

## Step 6: Tune HMC parameters with trial runs

Tuning HMC involves adjusting $M$, $\epsilon$, and $L$ for the particular application. The tuning process detailed in the previous section can be used as a guide to set the parameters.

Trial runs will have fewer samples than the full simulation, but should have sufficient number of samples to assess the acceptance rate and stationarity of the simulation. Note that the acceptance rate in trial runs may be higher than the rate during the full simulation. This can occur when the initial values are distant from the region of high posterior density.

One common approach in any MCMC is to allow a certain burn-in period before assessing the stationarity of the simulation. This period can typically be determined with trial trace plots. The required number of simulations in the burn-in period will typically be inversely related to the combined length defined by the tuning parameters. Higher values of $\epsilon$ and $L$ will move the chain quickly to the highest posterior region, while lower values will approach more slowly.

## Step 7: Run HMC

After the parameters have been tuned, a full HMC simulation can be run. Analysts often run multiple chains with different initial values. Most production software packages allow these chains to be run in parallel. For \textit{hmclearn}, parallelization can be accomplished using contributed R packages for the user's particular platform (e.g. Windows, mac OS, linux).

Once the simulation is complete, readily available diagonstics can be used to assess the convergence of the Markov chain. In R, the \textit{coda} package provides a number of standard diagnostics, such as the Geweke (1992) and Gelman and Rubin (1992) diagnostics. The \textit{bayesplot} package also provides graphical functions to assess convergence.

After the HMC simulation is complete and convergence has been assessed, the simulations can be used for statistical inference.

# HMC example

```{r, echo=TRUE}
head(warpbreaks)

summary(warpbreaks)
```

` ` `