# Hamiltonian Monte Carlo

MA538 - Bayesian Statistics: Final Project

*Romain Drai, Tin Huynh*

*5/4/2019*

## Introduction:

Sampling from posterior distributions using MCMC methods has become an integral part of Bayesian analysis. Posterior inference of Bayesian models is rarely tractable, and MCMC provides an asymptotically unbiased way to draw from the posterior target distribution by drawing a series of correlated samples. However, MCMC samplers are not created equal. Despite their popularity, random-walk Metropolis and Gibbs sampling are unable to efficiently explore target distributions with complex or high-dimensional spaces. This is because they explore the parameter space via 'guided' random walk.
Hamiltonian Monte Carlo (i.e. HMC) is an MCMC algorithm that avoids random walk behavior by using the first order gradient information and by simulating Hamiltonian dynamics. As a result, it converges to high-dimensional target distribution much more quickly than random-walk samplers. However, Hamiltonian MC is notoriously hard to execute. Its implementation necessitates the tuning of highly sensitive parameters: the number of steps L and the length of each step $\epsilon$. Our goal in writing this paper is twofold: we will explain how Hamiltonian MC works and how to easily implement it in STAN.

## Hamiltonian Monte Carlo

**Hamiltonian Monte Carlo (HMC):** is a Markov chain Monte Carlo (MCMC) method that uses the derivatives of the density function being sampled to generate efficient transitions spanning the posterior. HMC uses an approximate Hamiltonian dynamics simulation based on numerical integration which is then corrected by performing a Metropolis acceptance step. Thus, HMC allows the sampler to move much more quickly to and through the target distribution.
The goal of sampling with HMC is to draw from a density p($\theta$) for a parameter $\theta$ with the help of Hamiltonian dynamics and the Metropolis accept/reject step. In order to apply Hamiltonian dynamics, we need to introduce an "Auxiliary Momentum Variable" $\phi_j$ for each component $\theta_j$. Those two parameters are updated together in a Metropolis algorithm. Now, instead of sampling $p(\theta)$ we will sample $p(\theta, \phi)$. Since $\theta$ and $\phi$ are independent from each other, we can easily obtain $p(\theta)$ from $p(\theta, \phi)$ using $p(\theta, \phi) = p(\phi|\theta)p(\theta) = p(\phi)p(\theta)$.
In most applications of HMC, $\phi$ ~ multinomial(0,$\Sigma$), where $\Sigma$ acts as Euclidean metric to rotate and scale the target distribution. $\Sigma^{-1}$ is known as the mass matrix, and will be a unit, diagonal matrix.

### Hamiltonian Dynamic

**Hamiltonian Dynamics** consider the physical state of an object. Let $\theta$ and $\phi$ denote the object's position and momentum, respectively.The Hamiltonian of the object is defined as $H(\theta, \phi) = V(\theta) + T(\phi)$, where V is the potential energy and T is the kinetic energy. Hamiltonian dynamics describe the nature by which the momentum and position change through time.

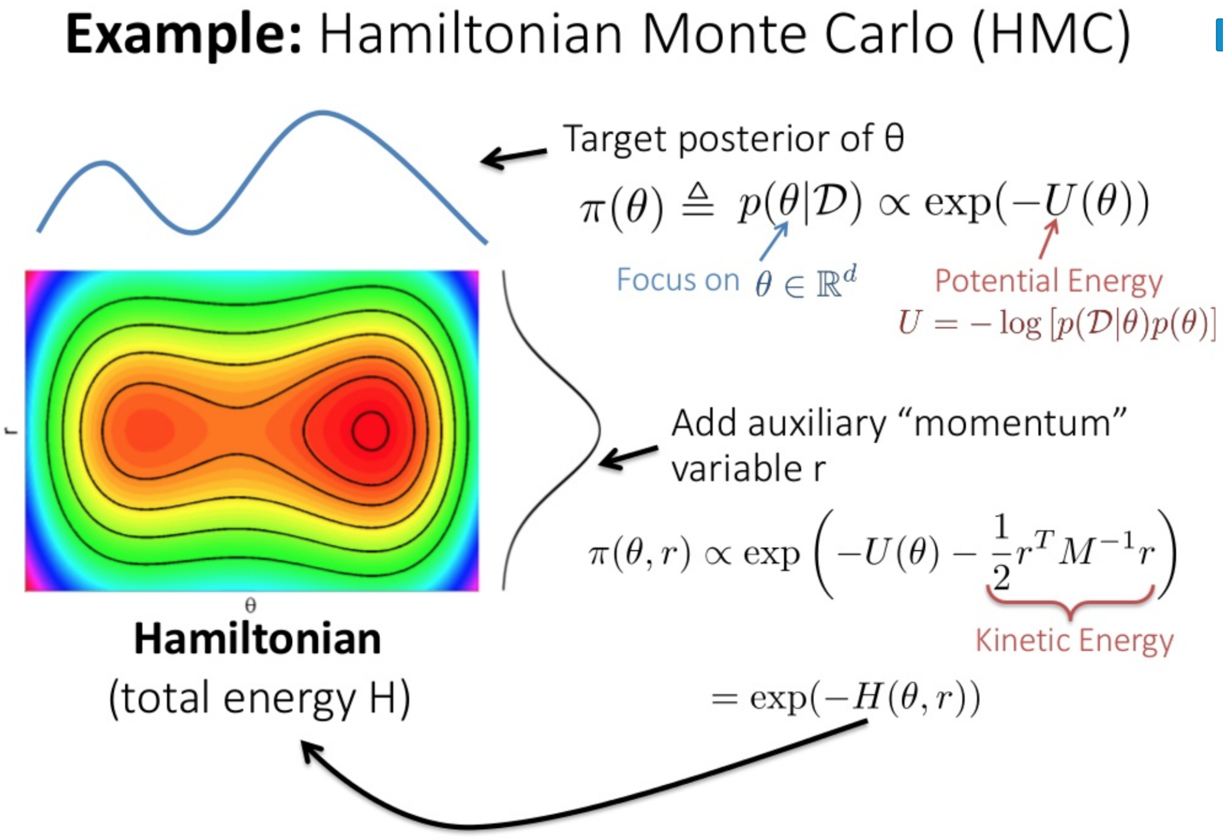
This movement is governed by the following system of differential equations called Hamilton's equations:

$$\frac{d\theta_i}{dt} = \frac{dH}{d\phi_i}$$
$$\frac{d\phi_i}{dt} = -\frac{dH}{d\theta_i}$$

In the MCMC applications of Hamiltonian Dynamics, we define $H(\theta,\phi) = -log(p(\theta,\phi)) = -log(\phi|\theta) - log(p(\theta)) = T(\phi|\theta) + V(\theta)$. We thus get:

$$\frac{d\theta_i}{dt} = \frac{dT}{d\phi_i}$$
$$\frac{d\phi_i}{dt} = -\frac{dV}{d\theta_i}$$

From this we can see that, the position will correspond to the variables of interest. The potential energy will be minus the log of the probability density for these variables.



**Example:** Hamiltonian Monte Carlo (HMC)
Target posterior of θ
$\pi(\theta) \triangleq p(\theta|\mathcal{D}) \propto \exp(-U(\theta))$
Focus on $\theta \in \mathbb{R}^d$
Potential Energy
$U = -\log[p(\mathcal{D}|\theta)p(\theta)]$
Add auxiliary "momentum" variable r
$\pi(\theta, r) \propto \exp\left(-U(\theta) - \underbrace{\frac{1}{2}r^T M^{-1} r}_{\text{Kinetic Energy}}\right)$
$= \exp(-H(\theta, r))$
θ
Hamiltonian
(total energy H)

The image describes the relationship between kinetic and potential energies. We can see that the contour lines separating the energy levels is reflected perfectly on target posterior distribution $p(\theta)$ and momentum distribution.

### Hamiltonian Dynamic Properties

Hamiltonian Dynamics are built on four important properties which are crucial to construct a well-balanced MCMC sampler.

**Reversibility:** This mean that for any state in time t we can move to t+s. This implies that we can simply

get back to state at time t by moving in the opposite direction. This is important because it would help to show that MCMC updates that use the dynamics leave the desired distribution invariant.
**Conservation of the Hamiltonian**: This property is necessary to show that we can use the Metropolis acceptance ratio to accept a proposal found using Hamiltonian dynamics. In practice, we can only make the Hamiltonian approximately invariant.
**Volume preservation** implies that we don't need to account for any change in volume in the acceptance probability for Metropolis updates. Without volume preservation we would need to compute the determinant of the jacobian for the mapping: a quantity that might be impossible to compute.
**Symplecticness** guarantees the jacobian will different from zero and that we are in a feasible region. It also implies volume preservation.

## Hamiltonian Monte Carlo Algorithm

To sample $(\theta, \phi)$ we need to find a way to approximate $\theta$ and $\phi$ at every step. We can apply Euler's rule to solve for $\theta$ and $\phi$ from the first order differential linear equation and it would help us to update $\theta$ and $\phi$

$$\phi_{t+\epsilon} = \phi_t + \epsilon \frac{d\phi}{dt}(t) = \phi_t - \epsilon \frac{dV}{d\theta}(\theta_t)$$
$$\theta_{t+\epsilon} = \theta_t + \epsilon \frac{d\theta}{dt}(t) = \theta_t + \epsilon \frac{\phi_t}{\sigma_{ii}}$$

Euler's rule works well when solving for $\theta, \phi$ at a specific time. But, it doesn't behave nicely when time keeps changing, that is why we need to slightly modify Euler's rule in order to sample more effectively. In this modified version, we update $\theta$ based on the updated of $\phi$ instead of the current $\phi$ . We called this algorithm the modified Euler's rule:

$$\phi_{t+\epsilon} = \phi_t - \epsilon \frac{dV}{d\theta}(\theta_t)$$
$$\theta_{t+\epsilon} = \theta_t + \epsilon \frac{\phi_{t+\epsilon}}{\sigma_{ii}}$$

Specifically in HMC, we will apply a version of modified Euler's rule called the **leapfrog method**. In this version we update $\phi$ with two half-steps rather than one full step. This modification allows us to use more efficiently the accept/reject step from the metropolis algorithm.
Here is the full HMC algorithm using the leapfrog method:

**Step 1:** Update $\phi$ with random draw from p($\phi$). Which is the same as prior distribution $\phi$~ N(0,Σ)

**Step 2:** Simultaneously update $(\theta, \phi)$ using L leapfrog steps:

- **(a)** $\phi = \phi + \frac{1}{2}\epsilon\frac{\partial \log p(\theta|y)}{\partial\theta}$
- **(b)** $\theta = \theta + \epsilon\Sigma^{-1}\phi$
- **(c)** $\phi = \phi + \frac{1}{2}\epsilon\frac{\partial \log p(\theta|y)}{\partial\phi}$

**Step 3:**

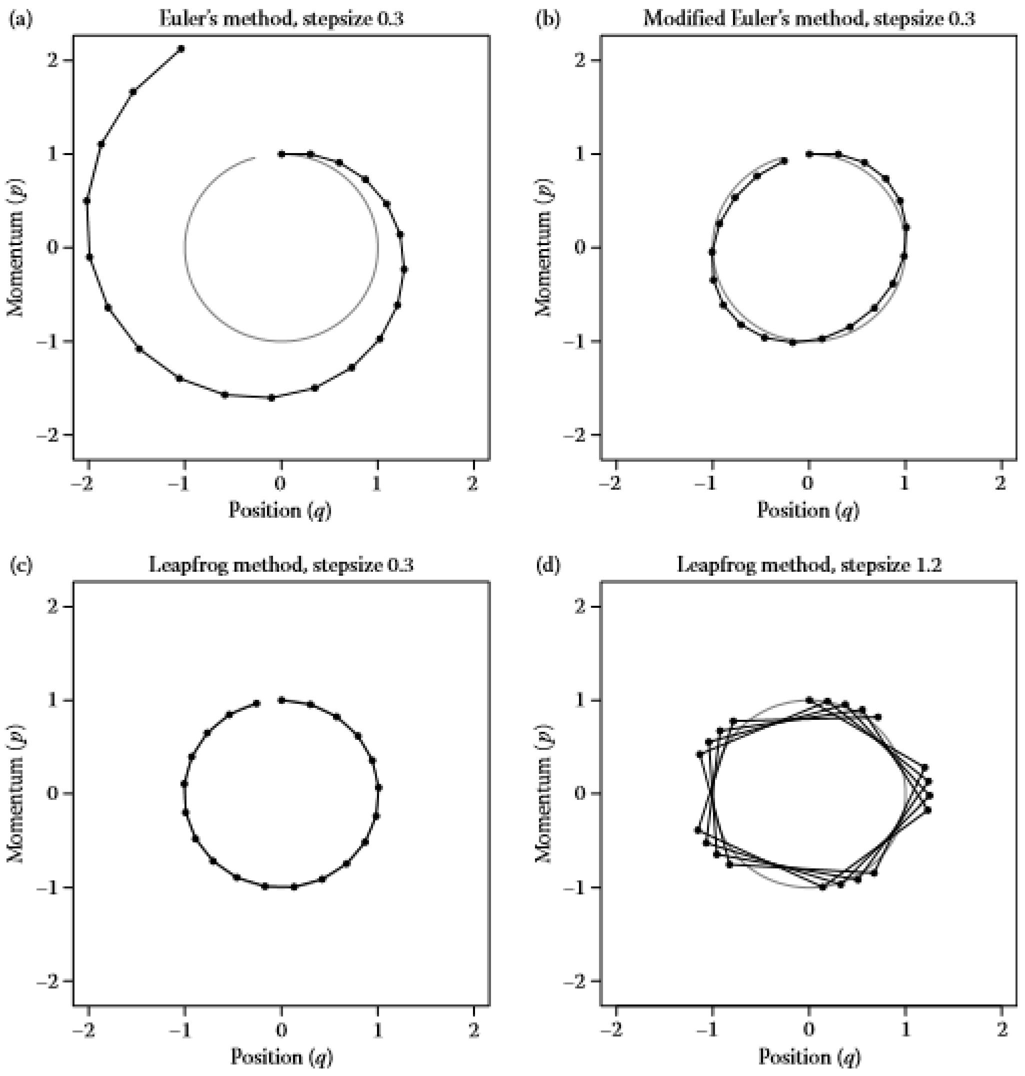


Figure 1: Euler's rule's Rule

$$
\begin{aligned}
r =& \frac{p(\theta^*|y)p(\phi^*)}{p(\theta^{t-1}|y)p(\phi^{t-1})} \\
\theta^t =& \theta^* \text{with probaility min(r,1)} \\
\theta^t =& \theta^{t-1} \text{ Otherwise}
\end{aligned}
$$

The image above compares the differences between three methods for approximating Hamiltonian dynamics .In this example, we have that: $H(\theta, \phi) = \frac{\theta^2}{2} + \frac{\phi^2}{2}$, which is represented by a gray circle. In part (a), we will apply Euler's rule's method, in part (b), we will apply modified Euler's rule's method. In part (c) and (d), we will apply leapfrog method. In each algorithm, we will set the inital $\theta = 0, \phi = 1$, $\epsilon$ = 0.3, and the steps in the simulated trajectory: L = 20. The only exception is in part (d), where we use $\epsilon = 1.2$.

So we can see that in part (a), the estimation for $H(\theta, \phi)$ is really close when it first starts but then become more and more inaccurate as the number of steps of simulation increases.

In part (b), the estimation is much better compared to part (a). This is because that we now use an updated $\phi$ to estimate a new $\theta$. There are still error but it starts to resemble the shape of our target distribution.

In part (c), the estimation is perfectly resembling the target distribution. This shows that at the same set up, leapfrog method would work better than Euler's rule and modified Euler's rule.

In part (d), when we change the $\epsilon$ to 1.2, we can see that all the point still be on the target distribution, however it is not evenly spread out. They group together and only reflect a small part out our target distribution instead of the whole target distribution. Therefore we need good parameter values in order to avoid getting biased samples.

## The Limitations of Hamiltonian Monte Carlo

Hamiltonian MC is one of the most efficient and accurate sampling algorithms, but it comes at a price. First it requires the gradient of the log-posterior. Computing this gradient function for complex model is tedious and many times even impossible. Secondly, the user needs to specify two highly sensitive parameters:

- L: the number of steps
- $\epsilon$: the length of each step

Unless the parameters are well-tuned, HMC algorithm will not perform optimally. If L is too large the algorithm will work too much for each iteration, but if it is too small the trajectory traced out in each iteration will be too short and sampling will devolve to a random walk. Likewise, if $\epsilon$ is too large the leapfrog interator will be inaccurate, but if $\epsilon$ is too small the simulation time will be too long. Additionally, the tuning of those parameters is also affected by the matrix $\Sigma$. If $\Sigma$ is poorly suited to the covariance, then the step size $\epsilon$ will have to decrease and the number of step L will have to increase.

Therefore, even a simplistic implementation of HMC requires extensise user input. This greatly impeded the widespread use of HMC, despite its efficiency. Fortunately, this changed with the release of the progamming language STAN, which automates all those tasks.

## STAN

STAN was initially released in 2012 and is a state-of-the-art probabilistic progamming language used for Bayesian inference. It is used primarily for modeling and high performance computation. It is written in C++ but can be accessed with interfaces in R, Python, Matlab and more. While STAN is primarily known for its implementation of MCMC samplers such as HMC and NUTS (an improved verison of HMC), it can also perform variational inference using the ADVI algorithm and optimization using the L-BFGS algorithm.

Using STAN greatly simplifies the implementation of HMC by limiting user intervention and costly tuning runs. First, it uses reverse mode automatic differentiation to calculate the gradient of the model. Secondly, it automatically tunes the parameters mentioned earlier. It tunes $\epsilon$ and L using **dual averaging** and also uses **Riemannian adaption** to locally adapt the mass matrix $\Sigma$ using the local curvature of $V(\theta)$.

Those two improvements to HMC allow the sampler to explore the distribution more efficiently and to remove user intervertion in parameter tuning. But the greatest contribution of STAN to HMC is its implementaion of the No U-Turn Sampler (NUTS). NUTS is an extension of HMC that removes the need to set the parameters L. It increases the computational efficiency of HMC by running the leapfrog trajectory until it turns around (i.e. U-turn).

## NUTS

NUTS algorithm was developed recently and was first published in 2014 by M.D. Hoffman and A. Gelman. NUTS is an extension of HMC in which L, the number of step parameter, is no longer fixed. We thus need a criterion to determine when the dynamics were simulated for long enough. We want to run the simulation until additional steps would no longer increase the distance between the initial $\theta$ and the proposal $\tilde{\theta}$ A convenient criterion would be to use the derivative of the squared distance with respect to time:

$$\frac{d}{dt}\frac{(\tilde{\theta}-\theta)\cdot(\tilde{\theta}-\theta)}{2} = (\tilde{\theta}-\theta)\cdot\frac{d}{dt}(\tilde{\theta}-\theta) = (\tilde{\theta}-\theta)\cdot\tilde{r}$$

We could then run the leapfrog steps until this quantity is less than zero. This would mean that the proposal $\tilde{\theta}$ is moving back toward $\theta$ (i.e. making a U-turn). However, this simple algorithm would not guarantee time reversibility and thus would not converge to the target distribution.
This problem can be solved by running simulations both forward and backward in time. Specifically, to achieve balance, NUTS will build a binary tree by using a doubling process. Using the leapfrog integrator, NUTS will create a path by running forwards or backwards 1 step, then forwards or backwards 2 steps, then forwards or backwards 4 steps and so on. This doubling process will then be halted until the forward or backwards sub-trajectory makes a U-turn and thus satisfies this condition:

$$(\tilde{\theta}-\theta)\cdot\tilde{r} < 0$$

The algorithm can be further improved by adding another stopping criterion. We can also stop the doubling process when the trajectory reaches a region of extremely low probability and satisfies this condition:

$$L(\theta) - \frac{1}{2}r\cdot r - \log(u) < -\Delta_{\max}$$

Where $\Delta_{\max}$ is a large value (e.g. 1000) and u, the slicing variable, is sampled from:

$$p(u|\theta, r) = \mathrm{Unif}(0, \exp\{L(\theta) - \frac{1}{2}r\cdot r\})$$

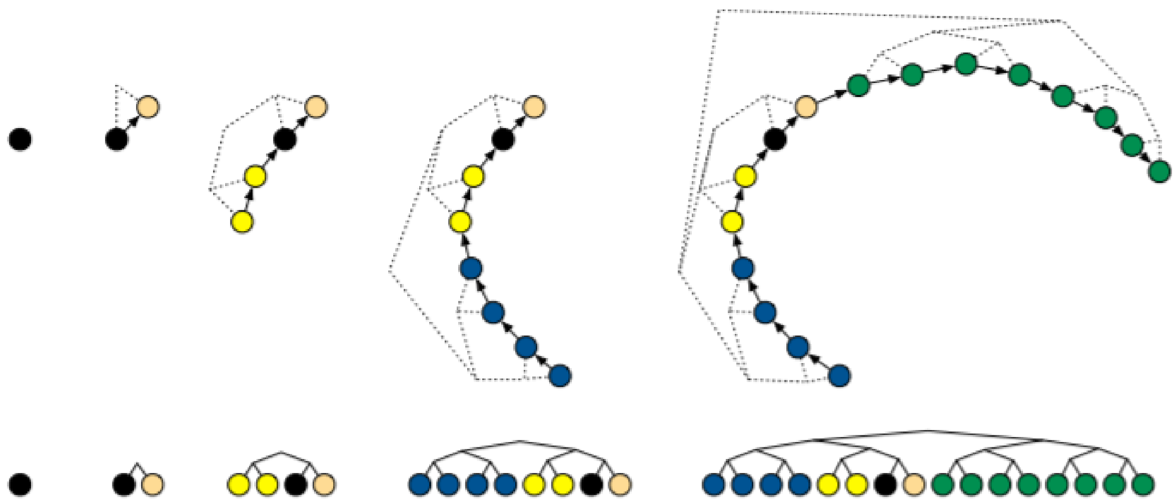


Figure 2: Building a binary tree using repeated doubling

However, even after this doubling forward/backward process, we are still not guaranteed to preserve the detailed balance. To achieve this balance, we would need to subset from the leapfrog position-momentum. Let:

- $\mathcal{B}$ be the set of all position-momentum states that the NUTS operator traces out during a given operator.
- $\mathcal{C}$ be the subset of those states to which we can transition without violating the detailed balance.

Specifically, B and C must satisfy the following conditions:

1. All elements of C must be chosen in a way that preserves volume. That is, any deterministic transformations of $(\theta, r)$ used to add a state $(\theta', r')$ to C must have a Jacobian with unit determinant.
2. $p((\theta, r) \in \mathcal{C} | \theta, r, u, \epsilon) = 1$
3. $p(u \leq \exp\{L(\theta') - \frac{1}{2} r' \cdot r'\} | (\theta', r') \in \mathcal{C}) = 1$
4. If $(\theta, r) \in \mathcal{C}$ and $(\theta', r') \in \mathcal{C}$ then for any $\mathcal{B}$, $p(\mathcal{B}, \mathcal{C} | \theta, r, u, \epsilon) = p(\mathcal{B}, \mathcal{C} | \theta', r', u, \epsilon)$

The first condition is automatically satisfied since the leapfrog steps are volume preserving. The second condition is satisfied as long as we include the initial state $\theta$, r in C. The third condition is satisfied if we exclude from C, states that satisfy: $u \leq \exp\{L(\theta') - \frac{1}{2} r' \cdot r' | (\theta', r') \in \mathcal{C}$. The fourth condition is satisfied as long as we exclude from C any states $\theta'$, r' that could not have generated B. For this last condition, we must consider two cases:

1. If the doubling procedure was stopped because the leftmost/rightmost state satisfied $(\tilde{\theta} - \theta) \cdot \tilde{r} < 0$, then the condition is automatically satisfied
2. If not, the condition is not satisfied and we must exclude from C all the states added during the last doubling iterations.

The figure below is an example of a trajectory generated during one iteration of NUTS. It displays which states are excluded from $\mathcal{C}$. The blue ellipse is a contour of the target distribution, the open circles along the trajectory are the elements of the set $\mathcal{B}$ with the black solid circle being the starting position. The blue arrow is the position vector and the magenta arrow is the momentum arrow at the final state of the trajectory. The doubling process stops here because the angle between the two arrows is greater than 90 degrees. The red solid circles are excluded from $\mathcal{C}$ because their probability is below the slice variable u and the states with a red cross were excluded from $\mathcal{C}$ in order to satisfy a detailed balance and satisfy condition 4.

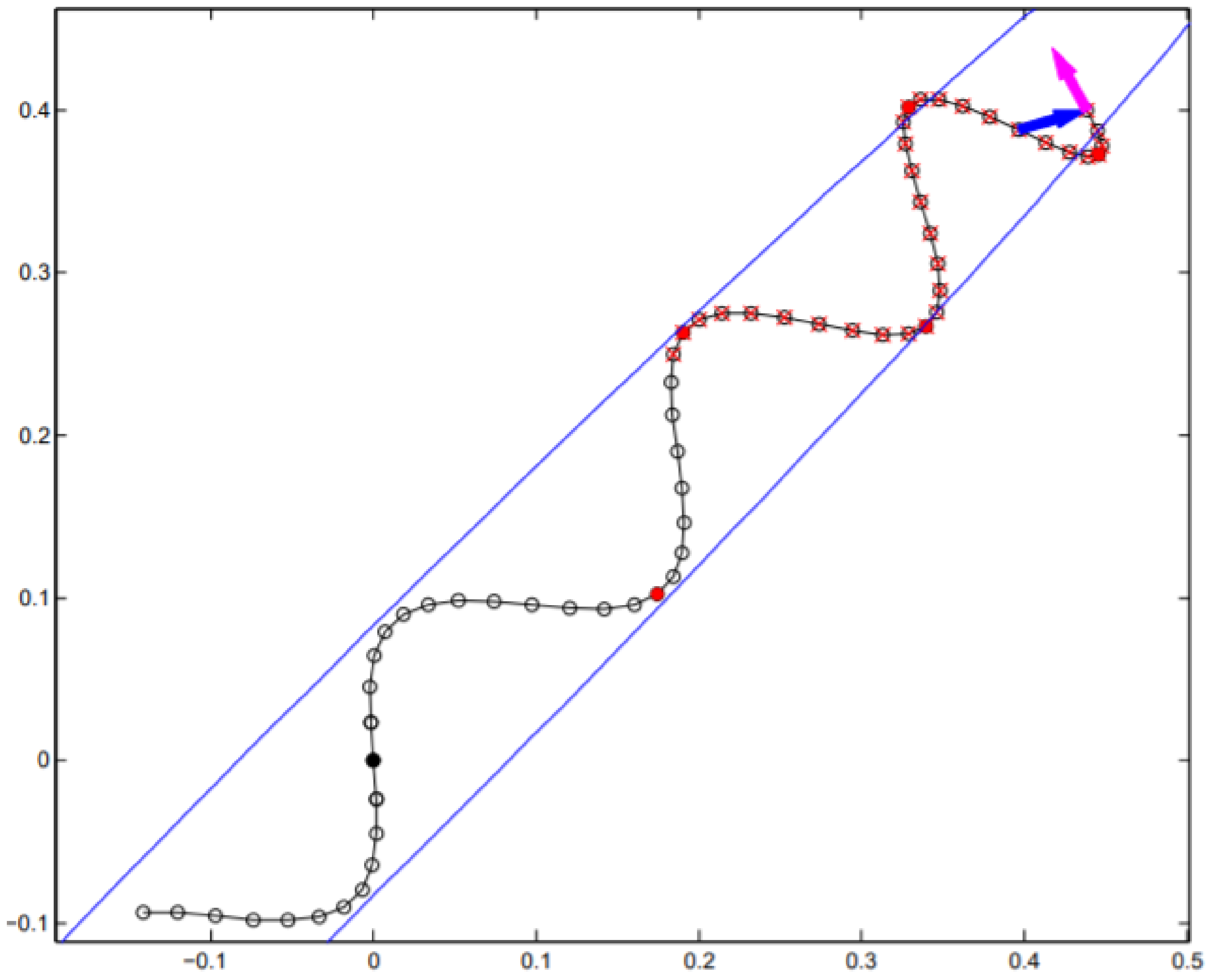


Figure 3: Trajectory generated during one iteration of NUTS

```
Given $\theta^0$, $\epsilon$, $\mathcal{L}$, $M$:
for $m = 1$ to $M$ do
   Resample $r^0 \sim \mathcal{N}(0, I)$.
   Resample $u \sim \text{Uniform}([0, \exp\{\mathcal{L}(\theta^{m-1} - \frac{1}{2}r^0 \cdot r^0\}])$
   Initialize $\theta^- = \theta^{m-1}$, $\theta^+ = \theta^{m-1}$, $r^- = r^0$, $r^+ = r^0$, $j = 0$, $\mathcal{C} = \{(\theta^{m-1}, r^0)\}$, $s = 1$.
   while $s = 1$ do
      Choose a direction $v_j \sim \text{Uniform}(\{-1, 1\})$.
      if $v_j = -1$ then
         $\theta^-, r^-, -, -, \mathcal{C}', s' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v_j, j, \epsilon)$.
      else
         $-, -, \theta^+, r^+, \mathcal{C}', s' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v_j, j, \epsilon)$.
      end if
      if $s' = 1$ then
         $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$.
      end if
      $s \leftarrow s'\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
      $j \leftarrow j + 1$.
   end while
   Sample $\theta^m, r$ uniformly at random from $\mathcal{C}$.
end for

function BuildTree($\theta, r, u, v, j, \epsilon$)
if $j = 0$ then
   Base case—take one leapfrog step in the direction $v$.
   $\theta', r' \leftarrow \text{Leapfrog}(\theta, r, v\epsilon)$.
   $\mathcal{C}' \leftarrow \begin{cases} \{(\theta', r')\} & \text{if } u \leq \exp\{\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r'\} \\ \emptyset & \text{else} \end{cases}$
   $s' \leftarrow \mathbb{I}[\mathcal{L}(\theta') - \frac{1}{2}r' \cdot r' > \log u - \Delta_{\max}]$.
   return $\theta', r', \theta', r', \mathcal{C}', s'$.
else
   Recursion—build the left and right subtrees.
   $\theta^-, r^-, \theta^+, r^+, \mathcal{C}', s' \leftarrow \text{BuildTree}(\theta, r, u, v, j - 1, \epsilon)$.
   if $v = -1$ then
      $\theta^-, r^-, -, -, \mathcal{C}'', s'' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v, j - 1, \epsilon)$.
   else
      $-, -, \theta^+, r^+, \mathcal{C}'', s'' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v, j - 1, \epsilon)$.
   end if
   $s' \leftarrow s's''\mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0]\mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
   $\mathcal{C}' \leftarrow \mathcal{C}' \cup \mathcal{C}''$.
   return $\theta^-, r^-, \theta^+, r^+, \mathcal{C}', s'$.
end if
```

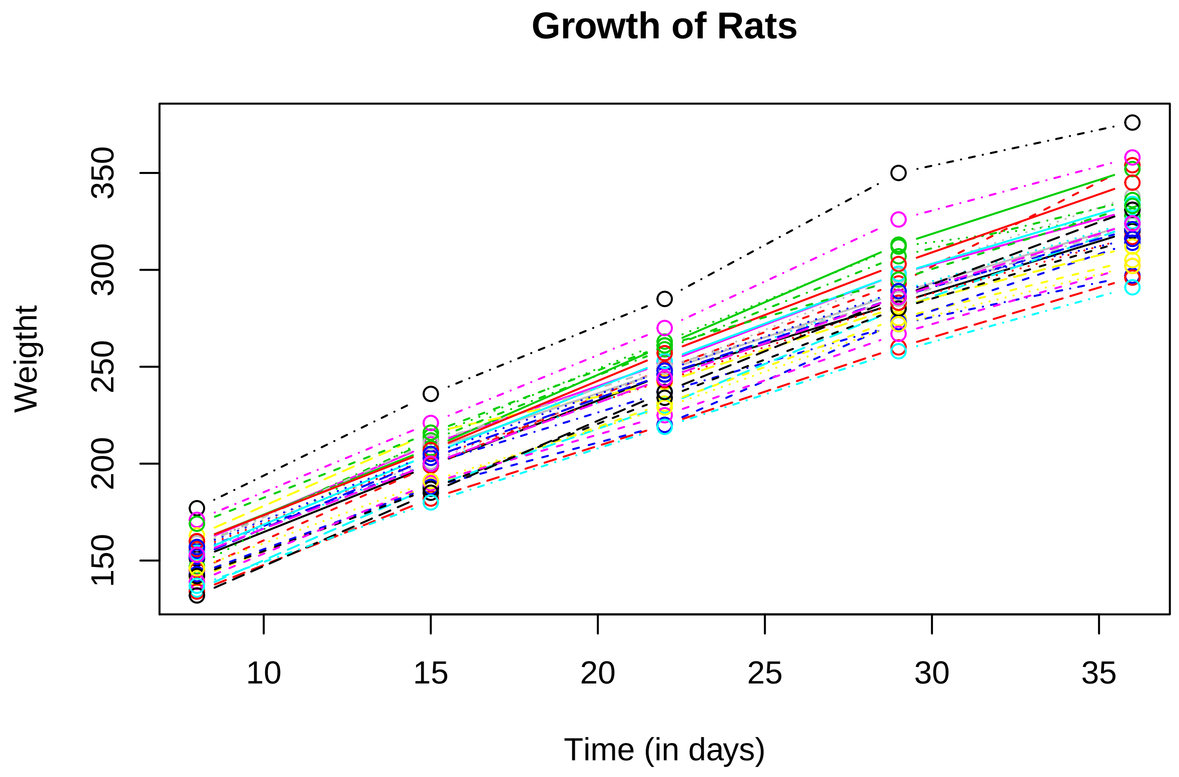
Figure 4: Naive version of NUTS. It can be improved be further optimized to reduce the number of computation and improve the transition kernels.

## Example: Rats Data

The example we will consider consists of modelling the growth of laboratory rats after their birth. The data consists of 30 rats that were weighted weekly for 5 weeks. The table below displays the weights of the first 15 rats:

| day8 | day15 | day22 | day29 | day36 |
|---|---|---|---|---|
| 151 | 199 | 246 | 283 | 320 |
| 145 | 199 | 249 | 293 | 354 |
| 147 | 214 | 263 | 312 | 328 |
| 155 | 200 | 237 | 272 | 297 |
| 135 | 188 | 230 | 280 | 323 |
| 159 | 210 | 252 | 298 | 331 |
| 141 | 189 | 231 | 275 | 305 |
| 159 | 201 | 248 | 297 | 338 |
| 177 | 236 | 285 | 350 | 376 |
| 134 | 182 | 220 | 260 | 296 |
| 160 | 208 | 261 | 313 | 352 |
| 143 | 188 | 220 | 273 | 314 |
| 154 | 200 | 244 | 289 | 325 |
| 171 | 221 | 270 | 326 | 358 |
| 163 | 216 | 242 | 281 | 312 |

The graph below shows the growth curve for each of the rats. The rats seem to experience a linear growth with a very small downward curvature.



## Normal Hierarchical Model

In our model, we will denote:

- $x_j$: age (in days)
- $Y_{ij}$: weight of $i^{th}$ rat at time $x_j$

In order to simplify our model, we will assume the rats' growth is linear and we will ignore the downward curvature of the data.
We will use the following Normal hierarchical model:

$$Y_{ij} \sim N(\alpha_i + \beta_i(x_j - \bar{x}), \sigma^2)$$

with priors:

- $\alpha_i \sim N(\mu_\alpha, \sigma_\alpha^2)$
- $\beta_i \sim N(\mu_\beta, \sigma_\beta^2)$

and the 'non-informative' hyperpriors:

- $\mu_\alpha \sim N(0, 100)$
- $\mu_\beta \sim N(0, 100)$,
- $\sigma_\alpha^2 \sim \Gamma^{-1}(0.001, 0.001)$
- $\sigma_\beta^2 \sim \Gamma^{-1}(0.001, 0.001)$
- $\sigma^2 \sim \Gamma^{-1}(0.001, 0.001)$

In this experiment we are primarily interested in predicting the average weight of the rats at birth. This would correspond to the intercept of the growth curve when $x_j = 0$. We denote the weight at birth as $\alpha_0$:

$$\alpha_0 = \mu_\alpha - \mu_\beta \bar{x}$$

We will now fit this model using STAN and R. We will sample from the posterior distributions using NUTS.

## STAN file

A STAN file is composed of 7 programming blocks:

- data
- transformed data
- parameters (required)
- transformed parameters
- functions
- model (required)
- generated quantities

The *data block* reads external information. In our case, STAN will read the data in our R document. The *parameters block* defines the sampling space. The *transformed parameter block* allows for parameter processing before the posterior is computed. This makes our code more efficient since it will reduce our memory usage. We input our model (i.e. likelihood and prior distributions) using the *model block*. We do not need to input the posterior distribution since STAN will automatically derive it. The *generated quantity block* allows for postprocessing variables. In our case, we will use it to compute $\alpha_0$, our variable of interest. We will also use this block to sample from the posterior predictive which is very useful for model checking.

```
data {
  int<lower=0> N;      // 2 primitive types
  int<lower=0> T;      // int and real
  real x[T];
  real y[N,T];
  real xbar;
}
parameters {
  real alpha[N];
  real beta[N];

  real mu_alpha;    // Prior
  real mu_beta;

  real<lower=0> sigmasq_y;
  real<lower=0> sigmasq_alpha;
  real<lower=0> sigmasq_beta;
}
transformed parameters {
  real<lower=0> sigma_y;
  real<lower=0> sigma_alpha;
  real<lower=0> sigma_beta;

  sigma_y = sqrt(sigmasq_y);
  sigma_alpha = sqrt(sigmasq_alpha);
  sigma_beta = sqrt(sigmasq_beta);
}
model {
  mu_alpha ~ normal(0, 100);
  mu_beta ~ normal(0, 100);
  sigmasq_y ~ inv_gamma(0.001, 0.001);
  sigmasq_alpha ~ inv_gamma(0.001, 0.001);
  sigmasq_beta ~ inv_gamma(0.001, 0.001);
  alpha ~ normal(mu_alpha, sigma_alpha); // vectorized
  beta ~ normal(mu_beta, sigma_beta);  // vectorized

  for (n in 1:N)
    for (t in 1:T)
      y[n,t] ~ normal(alpha[n] + beta[n] * (x[t] - xbar), sigma_y);

}
generated quantities {
  real alpha0;
  real y1_pred[T];

  alpha0 = mu_alpha - xbar * mu_beta;
  for (t in 1:T)
      y1_pred[t] = normal_rng(alpha[1] + beta[1] * (x[t] - xbar), sigma_y);
}
```

## R file

In the R file, we input the data and call the *stan* function to draw posterior samples.

```r
# Input data
y <- as.matrix(read.table('https://raw.github.com/wiki/stan-dev/rstan/rats.txt', header = TRUE))
x <- c(8, 15, 22, 29, 36)

rats_dat = list(y = y, x = x, xbar = mean(x), N <- nrow(y), T <- ncol(y))

# Fit model using STAN
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)
rats_fit <- stan('rats.stan', data = rats_dat,
                 chains = 4,
                 iter = 2000,
                 warmup = 1000,
                 thin = 1)
```

### HMC speed

The table below displays the time, in seconds, it took to run each chain. The warm-up phase took on average around 1.5 to 2 seconds, while the sampling phase took around 1 second for each chain. This shows the efficiency of the NUTS algorithm in STAN since we had to draw samples for 75 different variables.

```
##         warmup sample
## chain:1 11.137  1.072
## chain:2  1.862  1.254
## chain:3  1.481  1.619
## chain:4  1.834  1.183
```

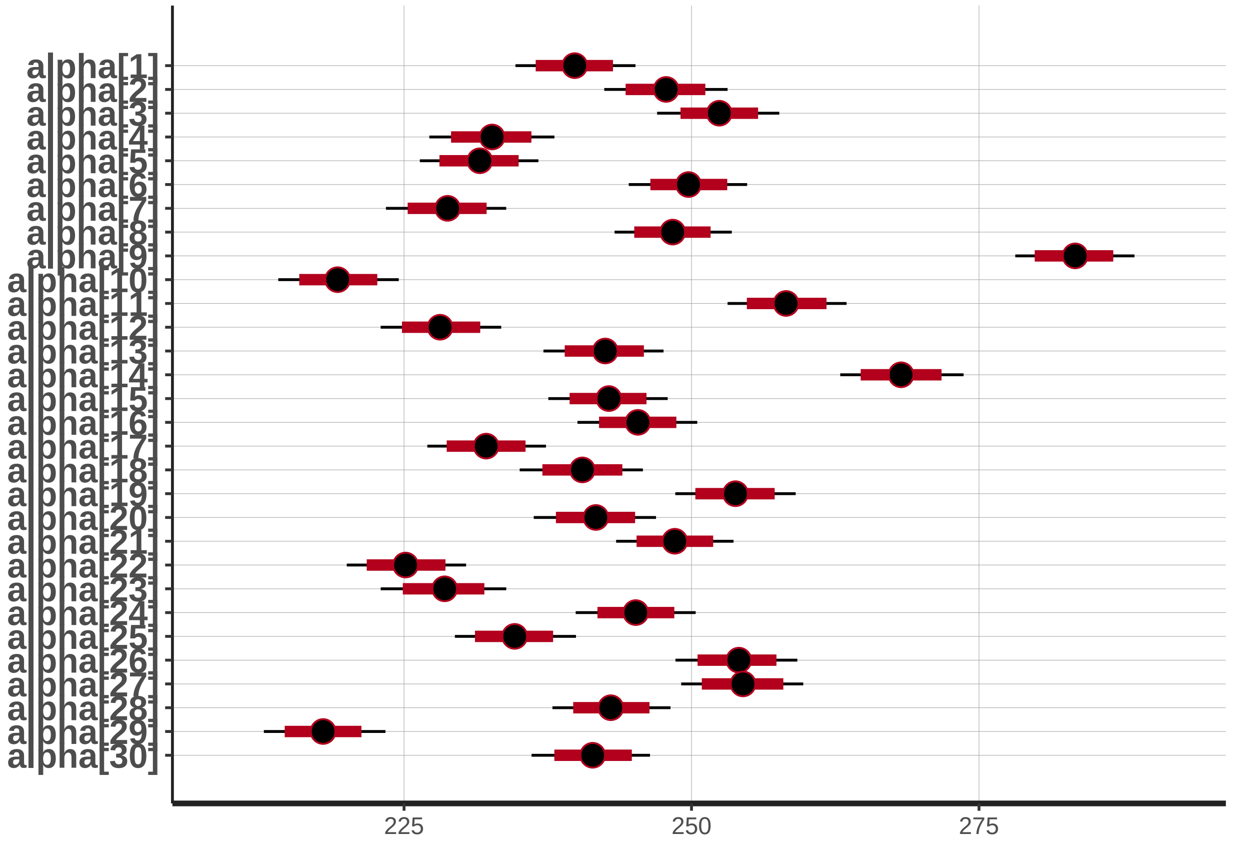
As one can see, the warmup phase runs systematically longer than the sampling phase even though it runs for less iterations. This is due to the fact that the HMC parameters are tuned during this phase.

**Summary Statistics**

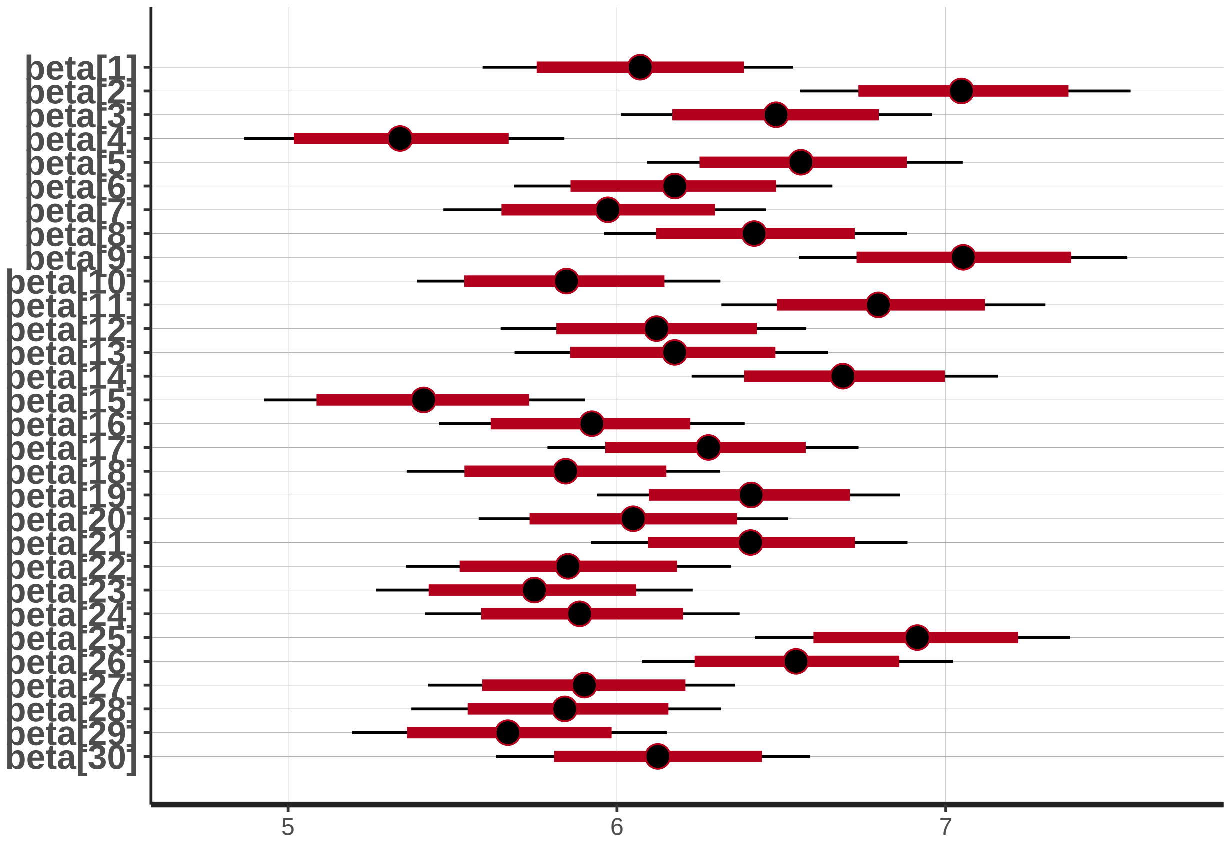
The table below summary statistics for each variable sampled. We can see that the Rhat value are always equal 1. This is a good sign since it indicates that the samples are well mixed.

```
## Inference for Stan model: rats.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##                  mean se_mean   sd    10%    50%    90% n_eff Rhat
## alpha[1]       239.85    0.04 2.65 236.45 239.84 243.17  4975    1
## alpha[2]       247.76    0.03 2.75 244.27 247.80 251.20  6201    1
## alpha[3]       252.41    0.03 2.67 249.04 252.42 255.79  6001    1
## alpha[4]       232.63    0.04 2.76 229.09 232.66 236.08  5654    1
## alpha[5]       231.56    0.04 2.68 228.08 231.59 234.96  5495    1
## alpha[6]       249.73    0.03 2.66 246.42 249.74 253.11  6785    1
## alpha[7]       228.74    0.03 2.68 225.31 228.79 232.18  6515    1
## alpha[8]       248.37    0.04 2.61 245.02 248.36 251.66  5295    1
## alpha[9]       283.30    0.04 2.67 279.84 283.36 286.68  4428    1
## alpha[10]      219.25    0.04 2.66 215.89 219.22 222.67  4678    1
## alpha[11]      258.26    0.04 2.67 254.81 258.22 261.74  4917    1
## alpha[12]      228.17    0.04 2.69 224.82 228.14 231.63  5080    1
## alpha[13]      242.46    0.04 2.69 238.97 242.50 245.86  5715    1
## alpha[14]      268.24    0.04 2.74 264.71 268.23 271.74  5122    1
## alpha[15]      242.78    0.04 2.63 239.40 242.81 246.08  5406    1
## alpha[16]      245.34    0.03 2.65 241.96 245.34 248.68  5732    1
## alpha[17]      232.16    0.03 2.64 228.71 232.14 235.56  6621    1
## alpha[18]      240.52    0.04 2.71 237.03 240.50 243.98  5532    1
## alpha[19]      253.81    0.03 2.71 250.33 253.82 257.23  6448    1
## alpha[20]      241.66    0.04 2.70 238.21 241.68 245.09  5857    1
## alpha[21]      248.56    0.03 2.60 245.22 248.54 251.88  5903    1
## alpha[22]      225.16    0.04 2.67 221.75 225.13 228.60  4851    1
## alpha[23]      228.50    0.04 2.78 224.89 228.53 231.98  5941    1
## alpha[24]      245.14    0.04 2.62 241.82 245.15 248.51  5284    1
## alpha[25]      234.60    0.03 2.70 231.17 234.62 237.97  6421    1
## alpha[26]      254.02    0.03 2.72 250.53 254.12 257.39  6834    1
## alpha[27]      254.46    0.04 2.74 250.90 254.48 257.98  5567    1
## alpha[28]      242.99    0.03 2.61 239.71 242.98 246.34  6162    1
## alpha[29]      217.96    0.04 2.66 214.63 217.96 221.29  4548    1
## alpha[30]      241.41    0.04 2.64 238.07 241.40 244.81  5633    1
## beta[1]          6.07    0.00 0.24   5.76   6.07   6.39  5465    1
## beta[2]          7.05    0.00 0.25   6.73   7.05   7.37  4547    1
## beta[3]          6.48    0.00 0.24   6.17   6.48   6.80  5767    1
## beta[4]          5.35    0.00 0.25   5.02   5.34   5.67  4953    1
## beta[5]          6.56    0.00 0.25   6.25   6.56   6.88  5366    1
## beta[6]          6.18    0.00 0.25   5.86   6.18   6.48  6046    1
## beta[7]          5.97    0.00 0.25   5.65   5.97   6.30  6296    1
## beta[8]          6.42    0.00 0.24   6.12   6.42   6.72  4963    1
## beta[9]          7.05    0.00 0.26   6.73   7.05   7.38  5212    1
## beta[10]         5.84    0.00 0.24   5.54   5.85   6.14  5431    1
## beta[11]         6.80    0.00 0.25   6.49   6.79   7.12  5376    1
## beta[12]         6.12    0.00 0.24   5.82   6.12   6.43  5346    1
## beta[13]         6.17    0.00 0.24   5.86   6.18   6.48  6096    1
## beta[14]         6.69    0.00 0.24   6.39   6.69   7.00  4979    1
## beta[15]         5.41    0.00  0.25    5.09    5.41    5.73  5043    1
## beta[16]         5.92    0.00  0.24    5.62    5.92    6.22  5429    1
## beta[17]         6.27    0.00  0.24    5.96    6.28    6.57  5745    1
## beta[18]         5.85    0.00  0.24    5.54    5.84    6.15  5966    1
## beta[19]         6.41    0.00  0.24    6.10    6.41    6.71  5249    1
## beta[20]         6.05    0.00  0.24    5.73    6.05    6.37  5419    1
## beta[21]         6.41    0.00  0.25    6.09    6.41    6.72  5596    1
## beta[22]         5.85    0.00  0.25    5.52    5.85    6.18  6022    1
## beta[23]         5.75    0.00  0.25    5.43    5.75    6.06  5354    1
## beta[24]         5.89    0.00  0.24    5.59    5.89    6.20  5832    1
## beta[25]         6.91    0.00  0.24    6.60    6.91    7.22  4652    1
## beta[26]         6.55    0.00  0.24    6.24    6.54    6.86  5533    1
## beta[27]         5.90    0.00  0.24    5.59    5.90    6.21  5650    1
## beta[28]         5.84    0.00  0.24    5.55    5.84    6.16  5738    1
## beta[29]         5.67    0.00  0.24    5.36    5.67    5.98  5032    1
## beta[30]         6.12    0.00  0.24    5.81    6.12    6.44  5052    1
## mu_alpha       242.46    0.04  2.69  239.06  242.46  245.84  4793    1
## mu_beta          6.18    0.00  0.11    6.05    6.18    6.32  4363    1
## sigmasq_y       37.10    0.13  5.73   30.42   36.44   44.41  1902    1
## sigmasq_alpha  218.57    1.04 66.12  148.36  207.20  304.32  4061    1
## sigmasq_beta     0.28    0.00  0.11    0.17    0.26    0.41  3405    1
## sigma_y          6.07    0.01  0.46    5.52    6.04    6.66  1891    1
## sigma_alpha     14.63    0.03  2.12   12.18   14.39   17.44  4259    1
## sigma_beta       0.52    0.00  0.10    0.41    0.51    0.64  3256    1
## alpha0         106.44    0.05  3.57  101.94  106.52  110.92  4924    1
## y1_pred[1]     154.99    0.12  7.39  145.60  154.92  164.58  3911    1
## y1_pred[2]     197.42    0.10  6.78  188.66  197.31  206.03  4257    1
## y1_pred[3]     239.89    0.11  6.64  231.44  239.88  248.37  3974    1
## y1_pred[4]     282.38    0.11  6.90  273.47  282.41  291.36  4037    1
## y1_pred[5]     324.93    0.11  7.34  315.44  324.87  334.34  4632    1
## lp__          -438.11    0.21  7.27 -447.71 -437.61 -429.26  1171    1
##
## Samples were drawn using NUTS(diag_e) at Mon Sep 09 13:45:59 2019.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

**Alpha estimates**

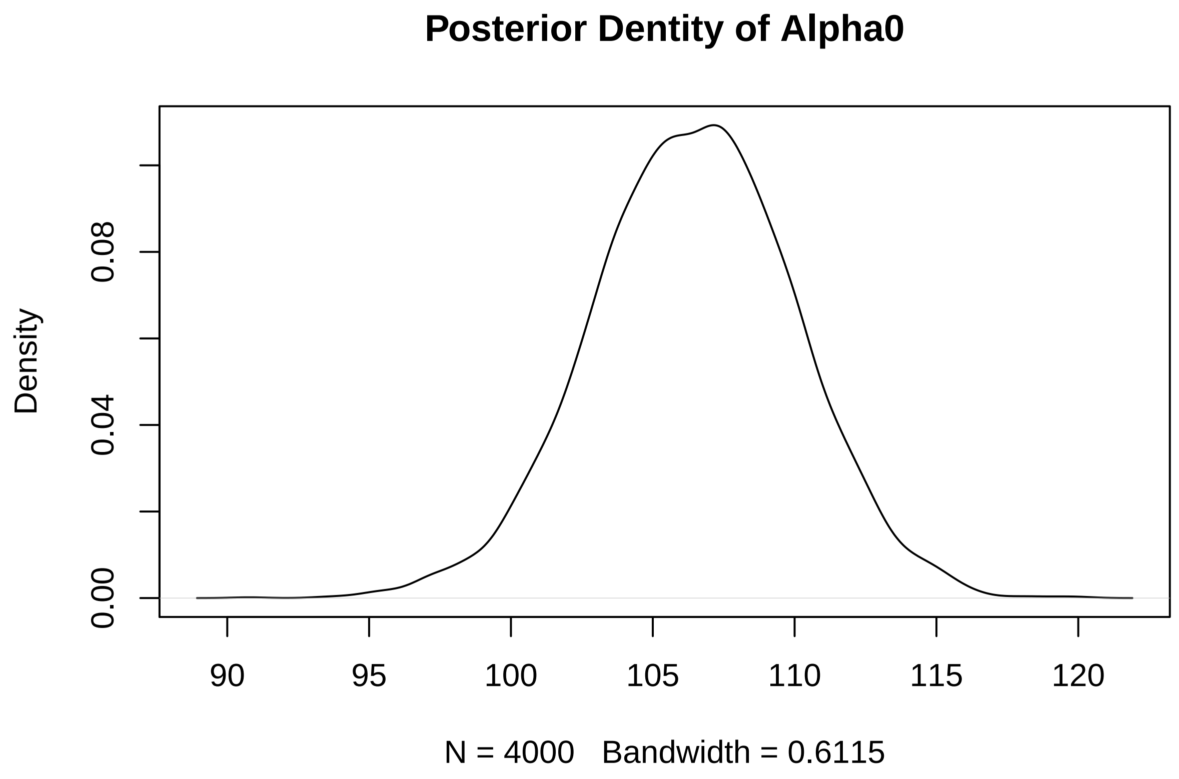


**Beta estimates**



**Weight at Birth**

The graph below displays the density plot of $\alpha_0$, the average weight of rats at birth:

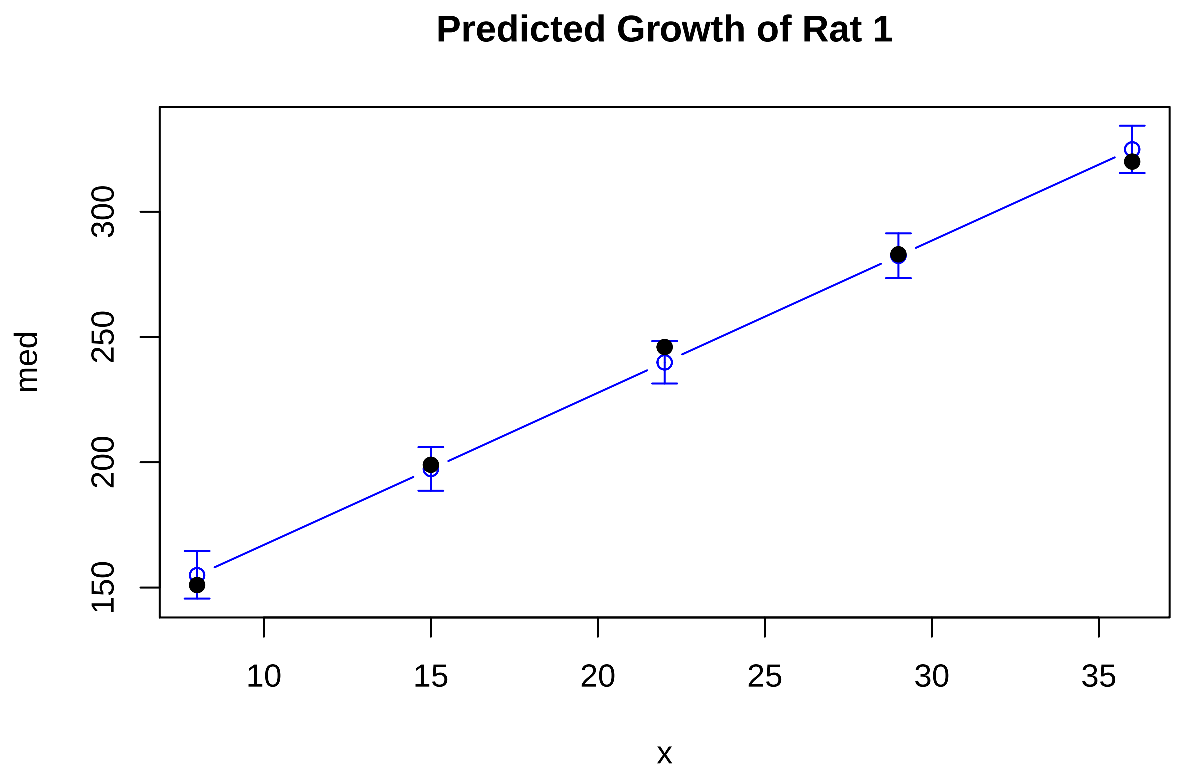


The MAP estimate of $\alpha_0$ is:

```
## [1] 107.1301
```

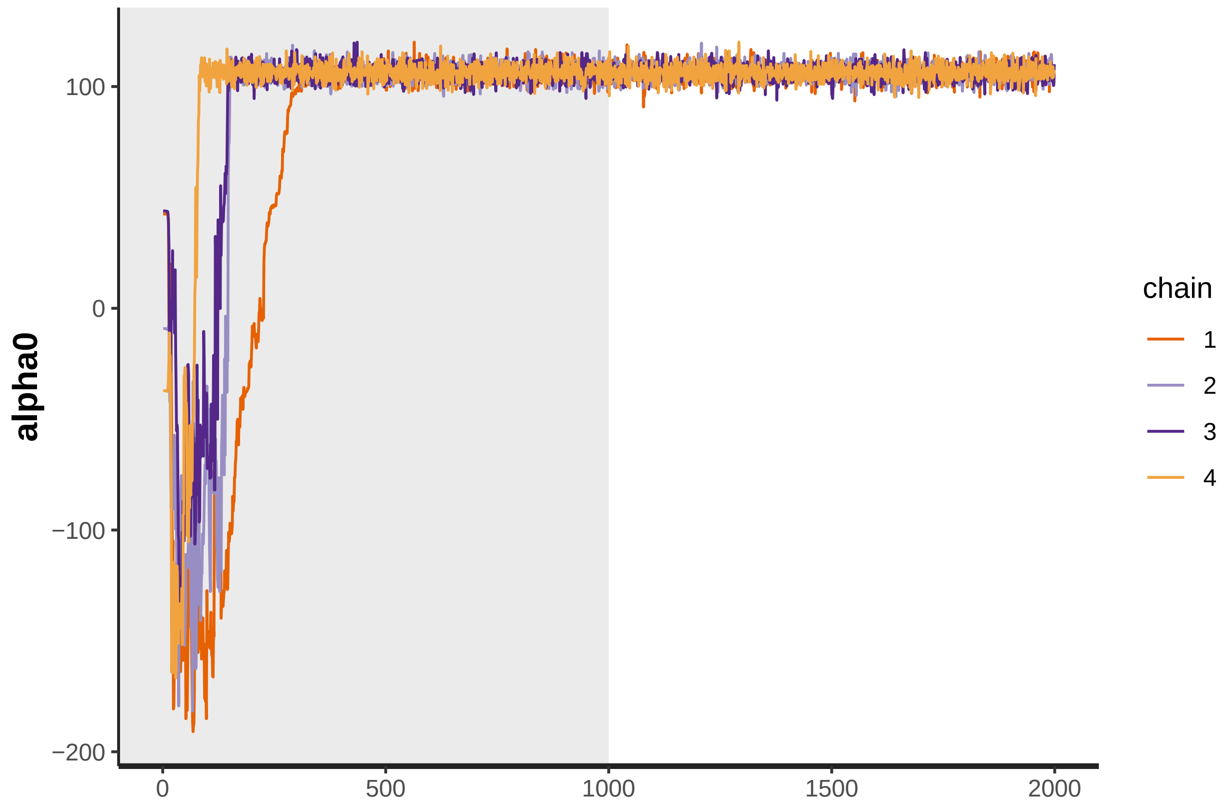
**Model Checking: Predictive Sample**

We can check our model fits the data well by using the predictive sample. The graph below compares the posterior estimates (in blue) and the original data points (in black). We can see that the model fits the data well since the data points are systematically within the 80% credible interval of the posterior sample.



### Trace plots

An important way to see if our sampler successfully reached the target distribution is to look at the trace plots. A good sampler should reach the target distribution before the end of the warm-up phase and should remain within this target distribution during the sampling phase. In the trace plot below it appears that each chain is successfully drawing from the target distribution.



### Why run Multiple Chains?

We run multiple chains for two reasons. The most obvious reason is to improve the speed of our algorithm. If we run each chain on a different core, our sampler will experience a linear speed-up. But the most important reason is to add robustness to our sample. Running multiple chains can help us spot bad convergence or bad mixing. This is especially useful when dealing with multimodal distribution. A good sampler should visit every mode of a distribution in a single chain. But, sometimes, a sampler might not have enough 'energy' to move from one mode to the next. In this case, different chains might get stuck in different modes and the trace plots will settle in different regions as shown in the image below.

This type of pattern in a trace plot in always a sign of bad mixing. Even if the chains end up visiting every mode, our sampler will still be biased because we will have no information on the weight the modes relative to each other.

## Conclusion

Today Hamiltonian Monte Carlo is one of the best samplers available. The fact that HMC uses a momentum function rather than random walk makes it far superior to Metropolis-Hasting and Gibbs sampling. This is

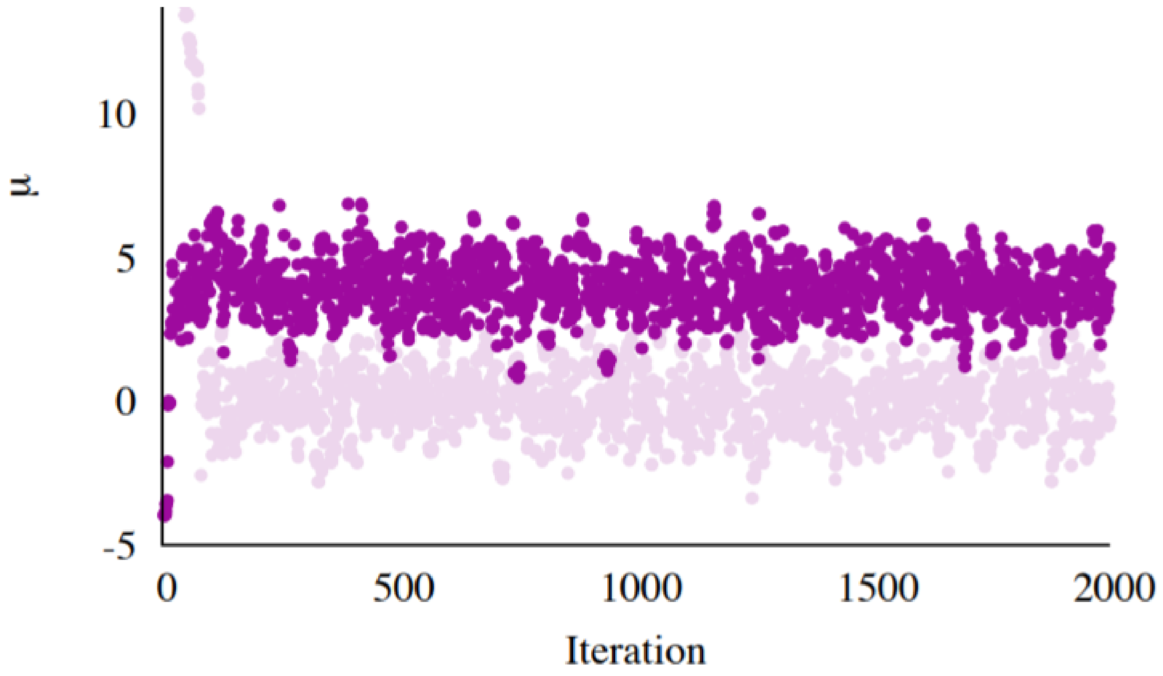


Figure 5: Bad Convergence

especially evident when exploring complex and high-dimensional distribution, since MH and Gibbs might not even converge to the target distribution. Moreover, HMC is computationally more efficient than its competitors. While a single iteration is more expensive, it has a higher acceptance rate and requires less iterations to obtain independent samples.
However, while Hamiltonian MC is one of the best sampler we have, it is not perfect and can still be improved. HMC still has difficulties with distributions with isolated modes and with distributions with very short or long tail. As a result, HMC is an actively researched topic. The STAN development team developed the NUTS algorithm as recently as 2014, and is still recruiting new developers to improve and implement new algorithms!

## Bibliography

1. Alex Rogozhnikov, *Hamiltonian Monte Carlo explained* http://arogozhnikov.github.io/2016/12/19/markov_chain_monte_carlo.html

2. Andrew Gelman and al., *Bayesian Data Analysis*

3. Matthew D. Hoffman and Andrew Gelman, *The No U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo* http://www.stat.columbia.edu/~gelman/research/published/nuts.pdf

4. Michael Betancourt, *Effecient Bayesian inference with Hamiltonian Monte Carlo* https://www.youtube.com/watch?v=pHsuIaPbNbY

5. Radford M. Neal, *MCMC using Hamiltonian dynamics* https://arxiv.org/pdf/1206.1901.pdf