HMC-MH: a comparison

Mirko Di Gangi, Leonardo Puricelli, Sangalli Chiara

Contents

Differences between Markov Chain Monte Carlo and the basic Monte Carlo method

Monte Carlo method

Statistical simulation is often used to approximate and sample features of distributions like means, standard errors and quantiles that cannot be computed in closed form or need to much time to be evaluated.

Whenever a function like an expected value or a likelihood need to be evaluated it's possible that the integral can't be computed analytically or its resolution process is long and time consuming, Monte Carlo method allows to approximate its value without having to solve it. MC method is a computational technique that

involves conducting a large number of random sampling experiments to simulate the behavior of stochastics systems and estimate unknown quantities.

This approach is based on the Law of Large Numbers (***LLN***): given n i.i.d random variables $X_1, ..., X_n$ with finite expected value $E(X_i) = \mu$ and positive variance $V(X_i) = \sigma^2 < \infty$, as n increases, their sample mean $\bar{X}$ converges in probability to μ:

$$\bar{X_n} = \frac{\sum_{i=1}^{n} X_i}{n} \to \mu$$

For instance, let X be a random variable with density function $f(X)$, and support χ, then its expected value can be computed as:

$$E_f[h(x)] = \int_{\chi} h(x) f(x) dx$$

This integral can be approximated with Monte Carlo, by generating a sample of length n, $(X_1, ..X_n)$, from the density f and than computing its empirical average:

$$\bar{h_n} = \frac{1}{n} \sum_{i=1}^{n} h(x_i)$$

By ***LLN*** the mean $\bar{h_n}$ converges to the true expected value $E_f[h(X)]$.

This is an example of a situation where the Monte Carlo method can be useful: the value of the integral $E_f[h(X)]$ is not direcly computed, instead it's approximated to the sample mean $\bar{h}$ of a sample obtained from f.

Markov Chain Monte Carlo

A Markov Chain is a sequence of values evolving over time with a probability of moving to the next state that depends only on the actual state. This process can be described in terms of a conditional probability $P_t(\theta^{(t+1)}|\theta^{(t)})$ defining to which value $\theta^{(t+1)}$ we are most likely to move from the actual $\theta^{(t)}$.

MCMC is a specific sampling Monte Carlo technique that aims to approximate or sample from a target density function f using a Markov Chain. This method becomes useful whenever the target distribution is complex or it's unfeasible to sample from it. By definition MCMC is any method producing a Markov Chain whose stationary distribution is π; meaning that, after a sufficient number of iterations, if $\theta^{(t)} \sim \pi$, then $\theta^{(t+1)} \sim \pi$. The core idea of MCMC is to construct a Markov chain that has the desired target distribution as its stationary distribution; by simulating the Markov chain the samples generated provide an approximation of the target distribution; π acts as a limiting distribution and once the MCMC sample converges to the target, it can be used to approximate any parameter of interest.

There are different algorithms that can be used to implement a MCMC procedure: the Metropolis-Hastings and the Hamiltonian are two possibilities.

Differences between MC and MCMC

The first difference between the Monte Carlo method and MCMC is that in the first one the samples are required to be independent, while the second approach is based on the creation of a Markov chain in which, by definition, each element depends on the previous one. As a consequence, MCMC requires a larger number of iteration to obtain a good approximation, because the dependencies within the sample slow down the convergence of the approximation to the true value.

However, Monte Carlo can't be implemented without a function to draw samples from; on the contrary, MCMC allows to obtain a sample approximately distributed as f without directly simulate from f itself.

To sum up, MCMC is slower but can be useful in a wider range of situation: no matter the initial value of the Markov chain, the transitions that the algorithm carries on will lead the chain to converge to a f, the target distribution from which it's impossible to sample directly.

Computational aspects of the Metropolis Hastings algorithm

Markov Chain Monte Carlo is a method that produces a Markov Chain that simulates the distribution of $p(\theta|y)$. The Metropolis-Hastings is one of the possible algorithms that performs this simulation; it can be splitted in 2 steps, the proposal and the correction. The first is any stochastic perturbation of the previous state of the chain, while the latter rejects any proposal that leads to a state which, compared to the previous one, is less likely to belong to the target density.

This algorithm is based on T iterations starting with a proposed value for θ, that comes from a specified distribution called proposal distribution.

Proposal distribution

The proposal distribution determines how new candidate values are generated during the simulation and at step t can be defined as follows:

$$q_t(\theta^{(*)}|\theta^{(t)})$$

Where θ^* is the proposed value and $\theta^{(t)}$ is the last value added to the chain.

For Metropolis-Hastings the proposal distribution can be non-symmetric, which means that it's possible to have $q_t(\theta^{(*)}|\theta^{(t)}) \neq q_t(\theta^{(t)}|\theta^{(*)})$.

By providing a set of rules for generating candidate values of θ, the proposal shapes the overall behavior of the algorithm. The choice of $q_t(\theta^{(*)}|\theta^{(t)})$ affects both the efficiency and the effectiveness of the algorithm: since it defines the region from which the MH proposes new values, the proposal influences the trajectory of the Markov chain.

A wide scope proposal allows for more extensive exploration of the parameter space, increasing the likelihood of finding better new values. However, this exploration comes at the cost of slower convergence, as the algorithm spends more time exploring low-probability regions. On the contrary, a narrow scope proposal distribution promotes faster convergence but can get the algorithm trapped in a local optima.

Moreover, the proposal must satisfy some properties, meaning that $q_t(\theta^{(*)}|\theta^{(t)})$ should be:

- Irreducible: given a starting point of the chain θ^a, the probability to reach another point θ^b in a finite number of moves must be different than 0.
- Aperiodic: a state θ^a cannot be visited only every k-th step.
- Recurrent: for every point θ^a must be ensured that, the Markov Chain, if it runs for a long enough time, eventually will go back to θ^a.

At stage t, the algorithm samples $\theta^{(*)}$ from the proposal distribution $q_t(\theta^{(*)}|\theta^{(t)})$. The proposed value depends only on the last value of the Markov Chain, $\theta^{(t)}$, and can either be accepted as the new value of the chain, becoming its $(t+1)$ element, or be rejected, setting $\theta^{(t+1)} = \theta^t$. Now we need to set up a rule to decide whether to accept or reject this proposed value.

Acceptance ratio

This acceptance/rejection rule is based on a ratio r that can be computed in the following way:

$$r = \frac{p(\theta^{(*)}|y)}{p(\theta^{(t)}|y)} \frac{q_t(\theta^{(t)}|\theta^{(*)})}{q_t(\theta^{(*)}|\theta^{(t)})}$$

r is made up of 2 different ratios: $\frac{p(\theta^{(*)}|y)}{p(\theta^{(t)}|y)}$ is a ratio of target densities, $\frac{q_t(\theta^{(t)}|\theta^{(*)})}{q_t(\theta^{(*)}|\theta^{(t)})}$ is a ratio of proposals, which is added as a correction term and it's due to the asymmetry of the proposal distribution. This means that, every time the algorithm is implemented with a symmetric proposal r will just be equal to the first ratio.

At the numerator the posterior is evaluated at the proposed value $\theta^{(*)}$, while at denominator is evaluated at $\theta^{(t)}$, the last value added to the chain; for the second ratio it's the opposite.

The probability of acceptance is the minimum between r and 1. This means that with probability $min(r, 1)$ $\theta^{(t+1)}$ will become the proposed value $\theta^{(*)}$ and with probability $1 - min(r, 1)$ $\theta^{(*)}$ will be rejected and the chain's $(t + 1)$ element will be equal to $\theta^{(t)}$.

Every time that r is higher than 1, the probability of accepting θ^* is equal to 1, so $\theta^t = \theta^*$. Whenever r is lower than 1 a way to decide whether to accept or reject the proposed value is to compare the acceptance ratio to u, a draw from $U \sim Unif(0, 1)$:

- If $r > u$ than $\theta^{(t+1)} = \theta^{(*)}$.
- If $r < u$ than $\theta^{(t+1)} = \theta^{(t)}$.

At this point a new value has been added to the chain and the process can restart from the beginning.

Structure of the MH algorithm

To sum up, the Metropolis-Hastings algorithm has the following structure:

For t = 1, . . . , T:

1. Given the last value of the chain, $\theta^{(t)}$ sample $\theta^{(*)}$ from the proposal distribution $q_t(\theta^{(*)}|\theta^{(t)})$
2. Calculate the acceptance ratio r:
$$r = \frac{p(\theta^{(*)}|y)}{p(\theta^{(t)}|y)} \frac{q_t(\theta^{(t)}|\theta^{(*)})}{q_t(\theta^{(*)}|\theta^{(t)})}$$
3. Set:
$$\theta^{(t+1)} = \begin{cases} \theta^{(*)} & \text{with probability min(r, 1)} \\ \theta^{(t)} & \text{otherwise} \end{cases}$$

This process is repeated for a high number of simulation T: the final output of the algorithm is a dependent sequence $(\theta^{(1)}, \ldots, \theta^{(T)})$ of values approximately drawn from the target distribution $p(\theta|y)$.

Very often this algorithm is implemented using a Gaussian distribution as the proposal, and for this reason the algorithm is known as random-walk Metropolis-Hastings; its biggest flaw is that it scales poorly whenever the target distribution gets more complex and when the number of dimension increases. In these cases the exploration of the Markov Chain will be extremely slow yielding to large autocorrelation and extremely imprecise estimators.

Hamiltonian Monte Carlo algorithm

Hamiltonian Monte Carlo allows the Markov Chain to explore the parameter space in a more efficient way than the Metropolis-Hastings. HMC includes momentum variables that suppress the random walk of the MH and allow each iteration to move smarter and faster in the parameter space. The goal of sampling is to draw from a density $p(\theta)$ for parameters θ, that is typically a Bayesian posterior $p(\theta|y)$.

Momentum variable and the Hamiltonian

For each component θ_j in the target space, Hamiltonian adds a momentum variable ϕ_j; both θ and ϕ are updated together in the algorithm, in which the proposal distribution for θ is largely influenced by ϕ.

In HMC the distribution $P(\theta)$ is augmented by an independent distribution $p(\phi)$, defining a joint density:

$$P(\theta, \phi) = P(\theta)P(\phi|\theta)$$

The distribution usually given to ϕ is a multivariate normal of dimension d $\phi \sim N_D(0, M)$ where M is a diagonal variance-covariance matrix, whose components are independent, so the j^{th} values of ϕ will be distributed as $\phi_j \sim N(0, M_{jj})$ $\forall j = 1, ..., d$. This distribution characterizes both the posterior and the prior for ϕ and from it will be drawn its updates.

The joint density $p(\phi, \theta)$ defines a Hamiltonian:

$$H(\phi, \theta) = -\log p(\phi, \theta) = -\log p(\phi|\theta) - \log p(\theta) = T(\phi|\theta) + V(\theta)$$

Where the terms $T(\phi|\theta)$ is called "kinetic energy" and $V(\theta)$ "potential energy".

From the joint distribution we are interested in sampling values of θ, while ϕ is added just to guarantee a faster convergence of the algorithm. The HMC requires also the gradient, a vector of partial derivatives of the log-posterior density. If θ has d dimension, the gradient is:

$$\frac{d \log p(\theta)}{d\theta} = (\frac{d \log p(\theta)}{d\theta_1}, ..., \frac{d \log p(\theta)}{d\theta_d})$$

Structure of the HMC algorithm

For $t = 1, \ldots, T$:

1. ϕ is updated with a random draw from its posterior $\phi \sim N(0, M)$.
2. ϕ and θ are simultaneously updated with a process that involves S leapfrog steps, each scaled by a factor ϵ. In each of these steps, both θ and ϕ changes, each one in relation with respect to the other, as it follows; for $s = 1, ...S$:
   (a) Use the gradient of the log-posterior distribution density of θ to make a half-step of ϕ:
   $$\phi_{\frac{s}{2}} = \phi + \frac{1}{2}\epsilon \frac{dlogP(\theta_s)}{d\theta_s}$$
   (b) Use the momentum vector to update the θ:
   $$\theta_s = \theta + \epsilon \; M^{-1}\phi_{\frac{s}{2}}$$

(c) Use again the gradient of θ to half-update ϕ:

$$\phi_s = \phi_{\frac{s}{2}} + \frac{1}{2}\epsilon \frac{dlogP(\theta_s)}{d\theta_s}$$

3. Label θ^{t-1} and ϕ^{t-1} the values of the parameter and momentum vectors at the start of the leapfrog process and θ^*, ϕ^* as the value after S steps.

4. Compute the ratio r:

$$r = \frac{P(\theta^{(*)})}{P(\theta^{(t)})} \frac{P(\phi^{(*)})}{P(\phi^{(t)})}$$

5. Set:

$$\theta^{(t+1)} = \begin{cases} \theta^{(*)} & \text{with probability min(r,1)} \\ \theta^{(t)} & \text{otherwise} \end{cases}$$

ϵ - the step size - is the time interval of each s step, when it tends to 0, the leapfrog step preserves the joint density $P(\theta, \phi)$. For finite ϵ, the joint distribution does not remain entirely constant, but it will slowly vary if ϵ is small.

Tuning the parameters

The tune parameters in the HMC algorithm, ϵ, the diagonal elements of M, and the number of leapfrog iterations S affect the behavior of the algorithm, so care is needed in choosing them. Wise choices may be the following:

- Consider M to be the identity matrix, so its diagonal elements are all 1.
- Set ϵ and S to guarantee that their product is equal to 1; for example we can set $\epsilon = 0.1$ and $S = 10$, so that $\epsilon * M = 0.1 * 10 = 1$.

At the beginning it's common practice to run some "warm up" iterations, to see how the parameters set behave, then modify them according to the results obtained and discard the warming up iterations.

Theory suggests that, for HMC to be optimally efficient, r must be $\approx 65\%$, hence it is recommended to adapt the algorithm to obtain an average acceptance ratio of 65%. If $r < 65\%$, then the leapfrog steps may be too ambitious, hence we should lower ϵ and increase S to have more iterations; on the other hand, if $r > 65\%$, the steps are probably too cautious, so we should proceed the opposite way by increasing ϵ and reducing S.

Sampling from a Banana distribution

In this example we implement both a Metropolis-Hasting and an Hamiltonian Monte Carlo algorithm to sample from the so called Banana Shaped Distribution, whose posterior density is proportional to:

$$\pi(\theta_1, \theta_2 | y_{1:n}) \propto p(\theta_1; \mu_1, \tau^2) p(\theta_2; \mu_2, \tau^2) \prod_{i=1}^{n} p(y_i; \theta_1 + \theta_2^2, \sigma^2)$$

In this example, we assumed $\mu_1 = \mu_2 = 0$, $\tau^2 = 1$, $\sigma^2 = 2$.

Figure 1 is a contour plot of the posterior density of the Banana distribution: the lighter the color the higher is the density; the dotted black lines are the level curves of the function.

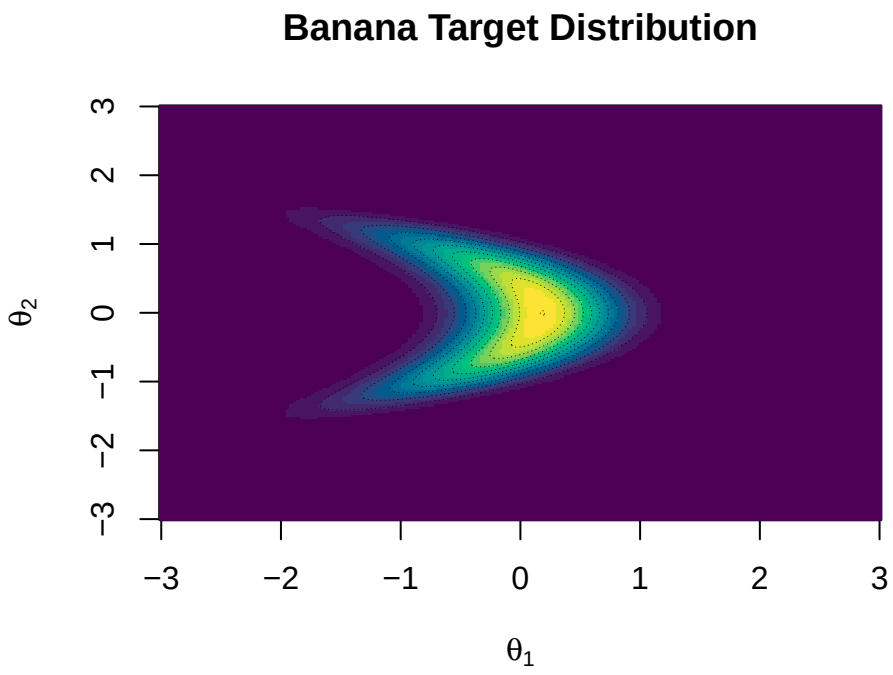

Figure 1: Posterior density of the Banana distribution

Implementation of the Metropolis-Hastings

The proposal we chose for θ_1 and θ_2 are both normal $N(\theta_i, 0.1)$. Since they are symmetric, we don't need the correction term $\frac{q(\theta^{(t)}|\theta)}{q(\theta|\theta^{(t)})}$ in the computation of r.

Since we are working with a two parameters distribution, we firstly have to update θ_1: draw θ_1^* from the proposal, compute $r_1 = \frac{p(\theta_1^{(*)}, \theta_2^{(t)})}{p(\theta_1^{(t)}, \theta_2^{(t)})}$, where t is the last available state and update $\theta_1^{(t+1)}$ according to the value of r_1. Then we repeat the same procedure for θ_2 using $\theta_1^{(t+1)}$ to compute r_2.

We set to 11000 the total number of iterations we want to run, 3000 of them will be used to warm up the algorithm and the remaining 8000 will be kept to approximate our parameters distributions. Since we are working with the logarithmic scale, the proposed values for θ will be accepted if $log(r)$ is bigger than $log(u)$ with u randomly sampled from a $Unif(0, 1)$.

The plots in Figure 2 show how our MH algorithm works in 4 different cases each with different initial values of θ_1 and θ_2: the green dots are the starting values, the red ones are the burn-in iterations, while the yellow ones are the 8000 iterations we kept to estimate the distribution. The first of these 4 samples will be later used to compare the MH with the Hamiltonian.

We implemented the diagnostic for the marginal distributions of θ_1 and θ_2, the results are shown in Figure 3, which contains the traceplot of the MH iterations, showing the convergence of the algorithm to the mode of the parameters, the acf plots showing the autocorrelation between the iterations for each lag and the comparison between the approximate distribution of the parameters and their real density. The traceplot doesn't show an optimal path of the convergence and the acf displays the presence of an autocorrelation problem.

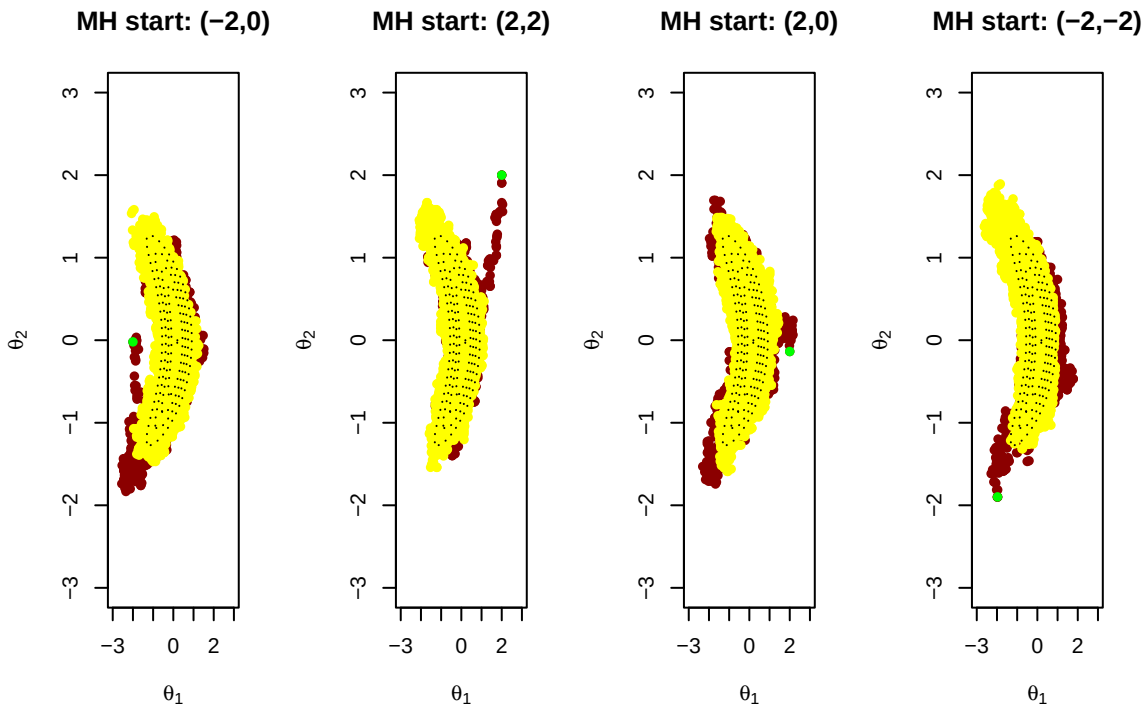

Figure 2: MH with different starts

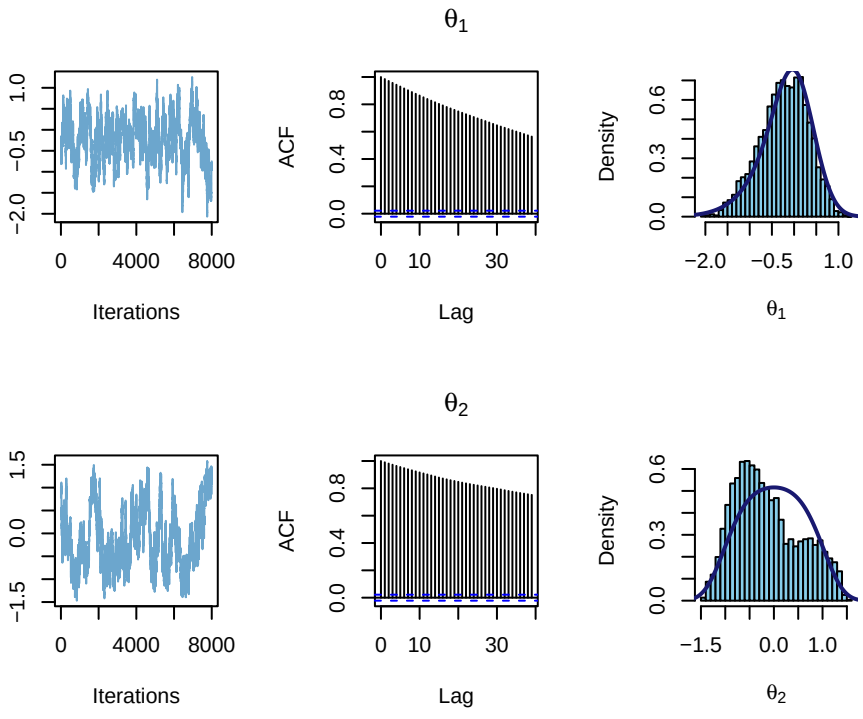

Figure 3: Traceplot, acf, marginal distribution of the parameteres - MH

Implementation of the Hamiltonian with STAN

STAN is a probabilistic programming language which provides full Bayesian inference for continuous-variable models through Markov Chain Monte Carlo methods such as the No-U-Turn sampler, an adaptive form of Hamiltonian Monte Carlo sampling. The No-U-Turn Sampler (NUTS) eliminates the need to set a number of steps and uses a recursive algorithm to build a set of likely candidate points that spans a wide range of the target distribution, stopping automatically when it starts to double back and retrace its steps. Empirically, NUTS performs at least efficiently as a well tuned standard HMC method.

Using the function ***sampling()*** we draw samples from the model defined in the *STAN file* "**IMPLEMENTATION.stan**". It computes 4 Markov chains, each time starting from different initial values of θ_1 and θ_2, that we set as: (-2,0), (2,0), (-2,-2), (2,2) and for each chain it discards half of the iterations considering them as warm-up.

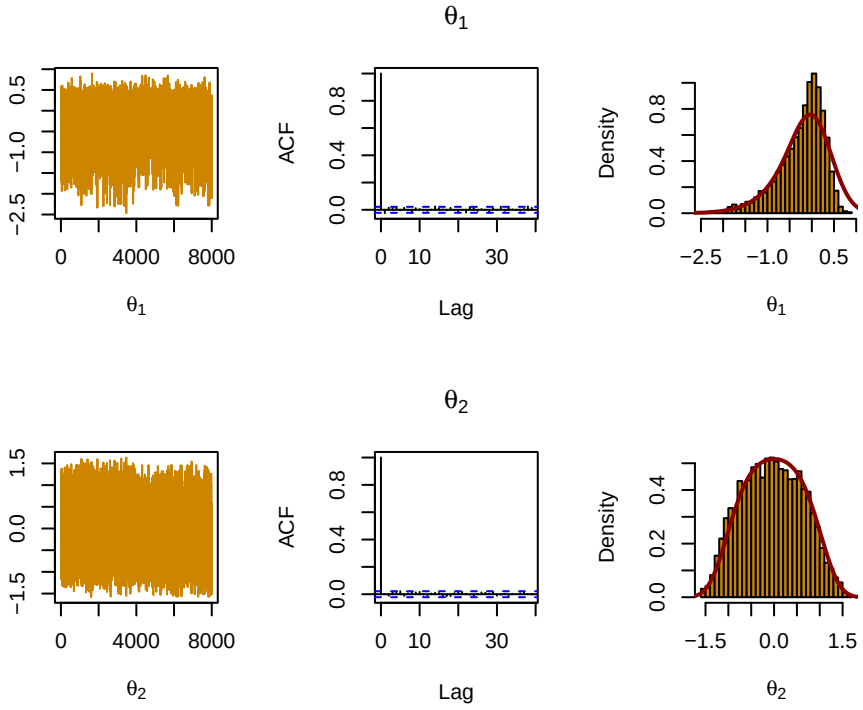

Figure 4: Traceplot, acf, marginal distribution of the parameteres - HMC

We repeat the diagnostic analysis and show the results in Figure 4, which displays traceplot, acf and marginal distribution of the parameters. The traceplot show a good path of the convergence, the autocorrelation plots don't show any problem and the densities seem quite close to the true.

Figure 5 shows all the sampled points of the target distribution estimated with STAN, colored according to the different chain they belong to and the sampled distribution computed with MH.

We can conclude that, even with a two parameters distribution, HMC approximate the posterior density in a better way than MH; this is clearly shown in Figure 3 and Figure 4: in the traceplots HMC explores the parameter space efficiently and the autocorrelations in the acf plot of the Hamiltonian are all close to zero.

References

- https://it.wikipedia.org/wiki/Pagina_principale;

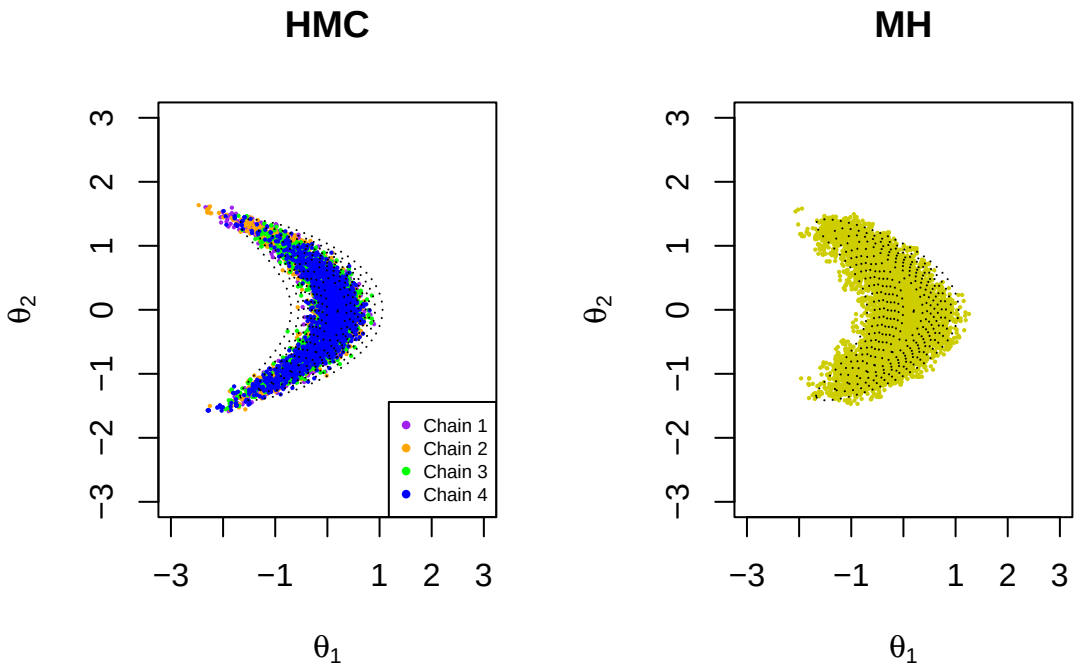

Figure 5: HMC and MH sampled distributions

- Slides of the course “Bayesian Modelling” by Federico Castelletti;
- “Bayesian Data Analysis”, Andrew Gelman, John Carlin, Hal Stern, Donald Rubin, David Dunson, and Aki Vehtari, 3rd edition, 2021;
- “Introducing Monte Carlo Methods with R”, C.P.Robert, G.Casella, 2010;
- https://mc-stan.org/users/interfaces/rstan;
- “A Conceptual Introduction to Hamiltonian Monte Carlo”, Michael Betancourt,2018.