Introduction

Lecturer: Riccardo Corradin

General info

Welcome to **Computational Statistics**

Before starting, I gratefully acknowledge **Tommaso Rigon**. Most of the material presented in this module is inspired (taken) by his former lecture notes and examples.

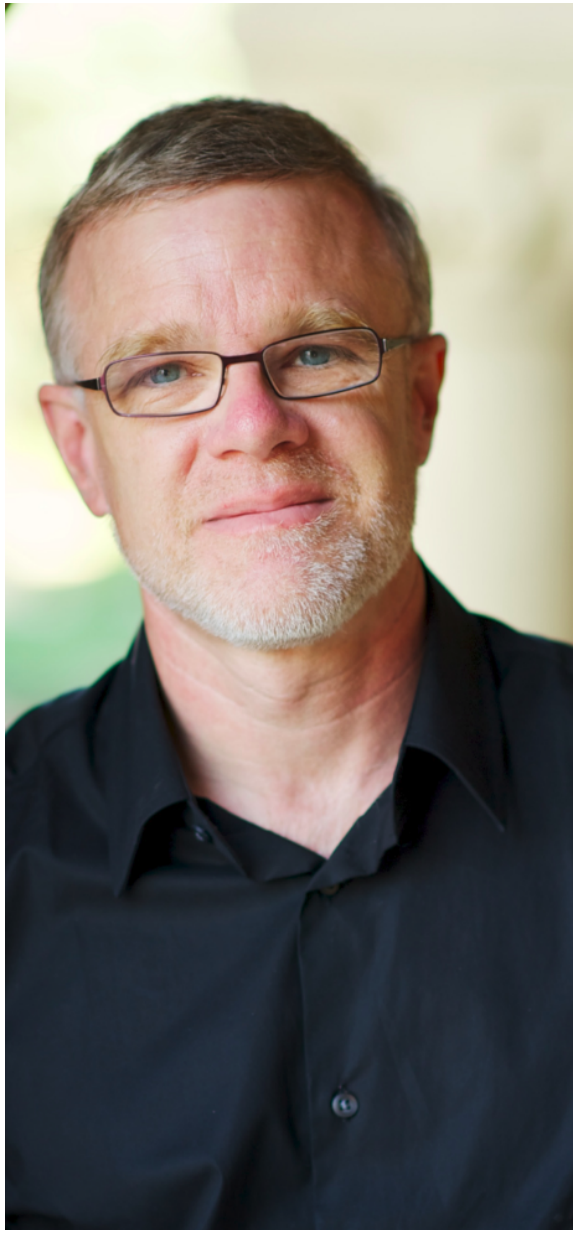
- This module is about **computational methods**
- Topics are divided in four **macro blocks**, namely
  - Metropolis Hastings and Gibbs sampling
  - Adaptive and dynamic-based methods
  - Importance-based methods
  - Approximate methods
- With all the methodologies, we will see also practical implementations
- All the material is available at

  link here

- You can report comments and typos at

  riccardo.corradin@unimib.it

Before starting



Inspired by the foundational talk of Art Owen at LMS Invited Lecture Series, CRISM Summer School (2018), in order you have to consider:

WORKING ORDER

1. Solutions from algebra
2. Solutions from calculus
3. Monte Carlo (MC) solutions
4. Approximate MC solutions

In the following lectures, we will address the last two points.

But remember! *"A big computer, a complex algorithm and a long time does not equal science."* Robert Gentleman.

Monte Carlo: why?

Bayesian inference in (less than) a nutshell

- We usually observe $\boldsymbol{X}^\intercal = (X_1, \ldots, X_n)$ **data**, where the generic X_i has support $(\mathbb{X}, \mathcal{X})$, here assumed to be **regular enough**. Each datum follows a shared distribution $f(x_i \mid \theta)$, indexed by an **unknown parameter** $\theta \in \Theta \subseteq \mathbb{R}^p$.
- The **empirical information is summarized** by the **likelihood** function

$$\mathrm{L}(\boldsymbol{X} \mid \theta) = \prod_{i=1}^{n} f(x_i \mid \theta).$$

- We usually set a **prior distribution** $\pi(\theta)$ for our unknown parameter θ.
- The core of our analysis is the posterior distribution resulting from a straightforward application of Bayes' theorem,

$$\pi(\theta \mid \boldsymbol{X}) = \frac{\mathrm{L}(\boldsymbol{X} \mid \theta)\pi(\theta)}{\int_\Theta \mathrm{L}(\boldsymbol{X} \mid \theta)\pi(\theta)\mathrm{d}\theta}.$$

- Except of few peculiar cases, the previous is not available in a closed form, as the **normalizing constant** (i.e. the integral in the denominator term) is often **intractable**.
  $\rightarrow$ no analytical solutions
- Numerical solution such as numerical integration of the **normalization constant** are highly unstable, especially in high dimensions or with multimodal distributions.

Bayesian inference in (less than) a nutshell

- The intuition beyond computational Monte Carlo methods for Bayesian inference is to produce a **sample** from the **posterior distribution** and then use such sample to perform posterior inference.

- If we can get random samples $\theta^{(1)}, \ldots, \theta^{(R)}$ from the **posterior distribution**, then we can **approximate** any (well posed) **functional** of interest as

$$\mathrm{E}[g(\theta) \mid \boldsymbol{X}] \approx \frac{1}{R} \sum_{r=1}^{R} g(\theta^{(r)}), \qquad \theta^{(r)} \sim \pi(\theta \mid \boldsymbol{X}),\ r = 1, \ldots, R.$$

- The previous is **justified** by the **law of large numbers**.

- We mainly distinguish among two approaches
  - When $\theta^{(1)}, \ldots, \theta^{(R)}$ are independent samples from $\pi(\theta \mid \boldsymbol{X})$, we refer to the approach as **Monte Carlo** method.
  - When $\theta^{(1)}, \ldots, \theta^{(R)}$ are dependent samples, and such dependence is driven by a Markov Chain, we follow a **Markov chain Monte Carlo** (MCMC) approach.

Review of Markov chains

Markov chains

- A sequence of **random elements** $Y^{(0)}, Y^{(1)}, \ldots, Y^{(R)}$, where the generic $Y^{(r)}$ has support $(\mathbb{Y}, \mathcal{Y})$, is a **Markov chain** if

$$\mathrm{P}\big(Y^{(r+1)} \in A \mid y^{(0)}, \ldots, y^{(r)}\big) = \mathrm{P}\big(Y^{(r+1)} \in A \mid y^{(r)}\big), \qquad \text{for any } A \subseteq \mathbb{Y}.$$

- **Dependence on the past** is fully driven by the **previous state** $Y^{(r)}$.
- The **conditional distribution** of $Y^{(r+1)} \mid y^{(0)}, \ldots, y^{(r)}$ is then the same of $Y^{(r+1)} \mid y^{(r)}$, and such distribution is called **transition kernel**.
- Given an initial condition $y^{(0)}$, a Markov chain is fully characterized by its transition kernel, which we assume does not depend on r (**homogeneity**).
  - $\rightarrow$ However, its parameters may vary over time.
- In continuous cases, the transition kernel is identified by a **conditional density** function, denoted with

$$\mathrm{k}(y^{(r+1)} \mid y^{(r)}).$$

- When the sample space is finite, the transition kernel is a **matrix**, say P.

A first example: AR(1)

- **Autoregressive processes** provides a simple illustration of **Markov Chains** on continuous state-space.
- Let $Y^{(0)} \sim N(30, 1)$ and let us define

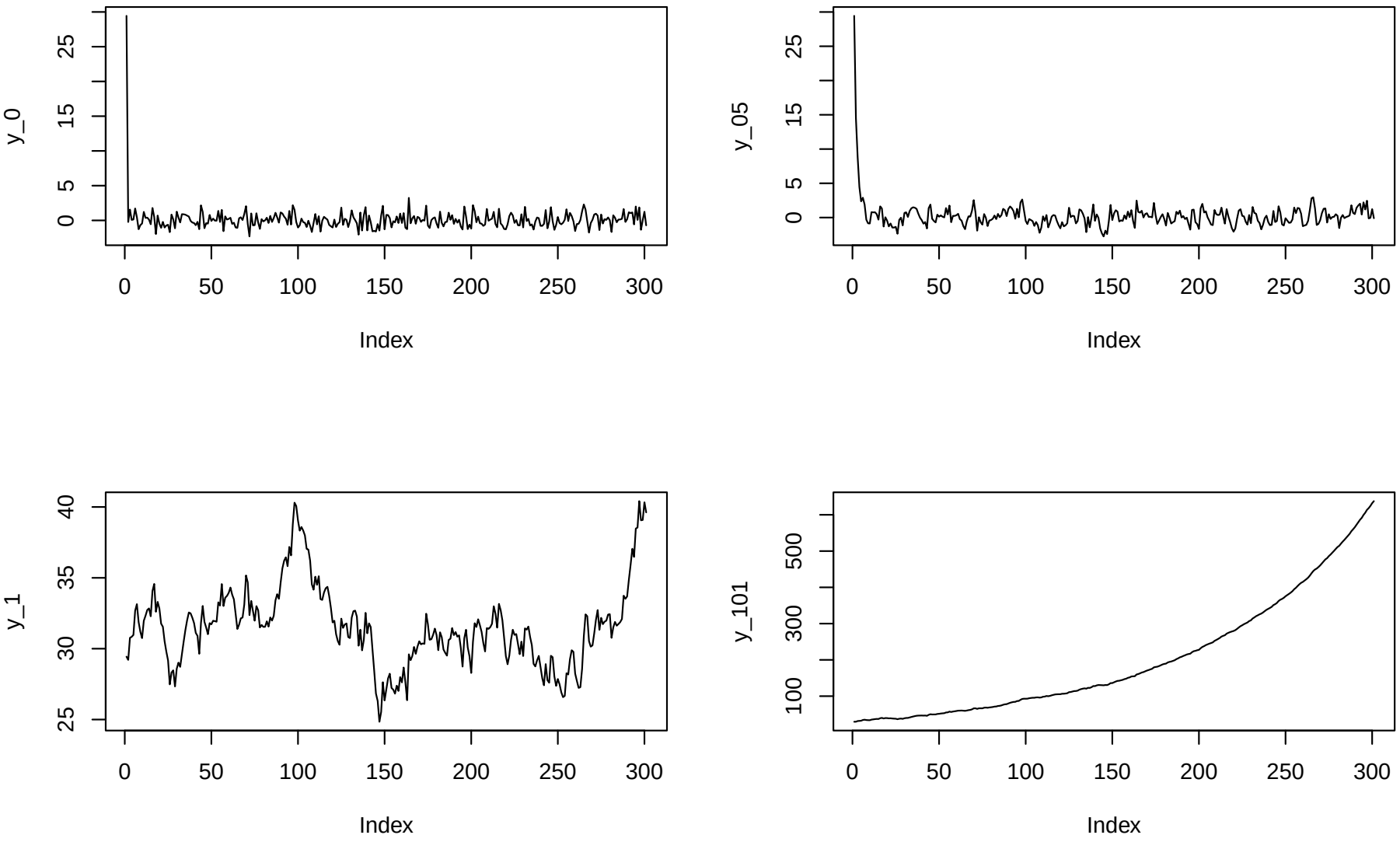
$$Y^{(r)} = \rho Y^{(r-1)} + \epsilon^{(r)}, \qquad \rho \in \mathbb{R},$$

  with the error terms $\epsilon^{(r)}$ being iid according to a $N(0, 1)$ distribution.
- The produced sequence $\{Y^{(r)}\}_{r \geq 0}$ is a first simple **example of Markov chain**.
- Thanks to the properties of the Gaussian distribution, it is also simple to write explicitly the **transition density** function

$$\left(y^{(r)} \mid y^{(r-1)}\right) \sim N\left(\rho y^{(r-1)}, 1\right).$$

- When the dependence parameter $|\rho| < 1$, the Markov chain has a more stable behavior.

A first example: AR(1)



Invariant distribution

- An increased level of **stability of a Markov chain** occurs when the latter admits an **invariant or stationary** probability distribution.
- A probability density $h(y)$ is invariant for a Markov chain with kernel k if

$$h(y^*) = \int \mathrm{k}(y^*, y) h(y) \mathrm{d}y,$$

  hence, a functional defined through the kernel **preserve the same distributional form** for $h(y)$.
- This is to say that the **marginal distributions** of $Y^{(r)}$ and $Y^{(r+1)}$ are the same and are equal to $g(y)$, since they are different just for the number of kernel actions, although $Y^{(r)}$ and $Y^{(r+1)}$ remain **dependent**.
- Roughly speaking, if a Markov chain admits a stationary distribution + some technical conditions, then for R large enough, the chain **"stabilizes"** around the invariant law.
- In the previous AR(1) example the stationary distribution is $N(0, 1/(1-\rho^2))$.

Invariant distribution

- Not every Markov chain admits a stationary law. However, Markov chains built for **Bayesian statistics** should always converge to an invariant distribution.
- Indeed, in Markov Chain Monte Carlo, the stationary distribution $h(y)$ represents the **target density** from which we wish to **simulate**, usually the posterior distribution in Bayesian inference.
- Then, we will make use of the following approximation

$$\int g(y)h(y)\mathrm{d}y \approx \frac{1}{R}\sum_{r=1}^{R} g(y^{(r)}),$$

  where $y^{(1)}, \ldots, y^{(R)}$ are generated according to a Markov chain, with $y^{(0)} \sim h(y)$.
- How to **construct a Markov chain** that converges to the **desired density** $g(y)$? Many possible strategies, depending on specific problems.
- Before delving into this key problem, let us briefly **review the assumptions** under which this approximation is reasonable.

Regularity conditions

- We will consider Markov chains that are **irreducible**, **aperiodic**, and **Harris recurrent**.
- A rigorous presentation of these properties is beyond the aims of this course, so we offer only a brief description in the **discrete case** to help the intuition.
- For a more detailed treatment, see Chapter 6 of Robert and Casella (2004).
- **Irreducibility**. The chain is irreducible if it does not "get stuck" in a local region of the sample space. In the discrete case, the chain is irreducible if all states are connected.
  - $\rightarrow$ As intuition, in the continuous case, this happens if the kernel is smooth and for each point we are mapping the entire support.
- **Aperiodicity**. The chain is aperiodic if it does not have any deterministic cycle.
- **Harris recurrent**. The chain is (Harris) recurrent if it visits any region of the sample space "sufficiently often".

Irreducibility

- The aforementioned properties are easy to formalize in the **discrete setting**, namely when the values of the Markov chain are $Y^{(r)} \in \{1, 2, ...\}$.
- The **first passage time** is the first r for which the chain is equal to j, namely:

$$\tau_j = \inf\{r \geq 1 : Y^{(r)} = j\},$$

  where by convention we let $\tau_j = \infty$ if $Y^{(r)} \neq j$ for every $r \geq 1$.
- Moreover, let us denote the **probability of return** to j in a finite number of step, starting from j'

$$\mathrm{P}(\tau_j < \infty \mid y^{(0)} = j').$$

- Hence, the chain is **irreducible** if $\mathrm{P}(\tau_j < \infty \mid y^{(0)} = j') > 0$ for all $j, j' \in \mathbb{N}$.

Aperiodicity

- Consider the two-state chain with **transition matrix**

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

- With the previous matrix, if we have two states, say 1 and 2, the Markov chain induced by P is **alternating those two states**

$$P\boldsymbol{v}^r = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \boldsymbol{v}^{r+1}, \qquad \text{and} \qquad P\boldsymbol{v}^r = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \boldsymbol{v}^{r+1},$$

- The two-step ahead transition matrix is $P^2 = I$, so $P^{2r} = I$ and $P^{2r+1} = P$ for all $r \geq 1$.
- Hence, due to periodicity this chain is **failing to converge anywhere**.
- In the discrete case, we call a state j aperiodic if the set

$$\{r \geq 1 : [P^r]_{jj} > 0\}$$

  has no common divisor other than 1.
- A chain is **aperiodic** if all its states are aperiodic

Harris recurrence

- Informally, a state j of an **irreducible Markov chain** is recurrent when it is (expected to be) visited by the chain "infinitely often", i.e.

$$\mathrm{E}[\eta_j] = \infty, \qquad \text{where} \qquad \eta_j = \sum_{r \geq 1} \mathbb{I}_{[Y^{(r)} = j]}.$$

- More formally, in the discrete setting a state $j \in \mathbb{N}$ is recurrent if and only if

$$\mathrm{P}(\tau_j < \infty \mid y^{(0)} = j) = P(Y^{(r)} = j \text{ for infinitely many } r \mid y^{(0)} = j) = 1.$$

- The above definition, with the necessary adjustments, is a **sufficient condition** for recurrence in the continuous case.

- Indeed, in the continuous case recurrence is defined in terms of the **average number of passages** on a Borel set, which must be divergent.

- The stronger **Harris recurrence** condition is mostly needed to fix measure-theoretic pathologies.

Invariant measure

- A Markov chain that is aperiodic and Harris recurrent displays a quite stable behavior, so one may wonder if it admits an **invariant distribution**.
- In general, the answer is no: the Gaussian random walk is an example.
- Indeed, we call **Harris positive** a Markov chain, which is Harris recurrent and admits an invariant probability distribution.
- In the discrete case, this occurs if and only if $\mathrm{E}(\tau_j \mid y^{(0)} = j) < \infty$.
- However, something can be said about the existence of invariant measures in general.

Theorem

If $\{Y^{(r)}\}_{r\geq 1}$ is a recurrent chain, there exists an invariant σ-finite measure which is unique up to a multiplicative factor.

- Unfortunately, such an invariant measure is not necessarily a probability measure!

Reversibility and detailed balance

- What follows is a popular **sufficient condition** to ensure a recurrent chain is also **positive recurrent**. That is, it admits an invariant probability distribution.
- Interestingly enough, such a condition also has a quite intuitive interpretation.
- We call a Markov chain $\{Y^{(r)}\}_{r\geq 1}$ **reversible** if the distribution of $Y^{(r)}$ conditionally on $Y^{(r+1)}$ is the same as the distribution of $Y^{(r+1)}$ conditionally on $Y^{(r)}$.
- A Markov chain $\{Y^{(r)}\}_{r\geq 1}$ with transition kernel k satisfies the **detailed balance condition** if there exists a function such that

$$\mathrm{k}(y \mid y^*)h(y) = \mathrm{k}(y^* \mid y)h(y^*).$$

Theorem

If $\{Y^{(r)}\}_{r\geq 1}$ satisfies the detailed balance condition with h a probability density function, then h is the invariant (stationary) density, and the chain is reversible.

Convergence to equilibrium

- From now on, we will always assume the **aperiodicity** and **Harris positivity** properties, assuming the existence of a stationary probability density h.
- The following result establishes that a chain converges in total variation to its invariant measures as $r \to \infty$.
- Importantly, this occurs regardless the initial conditions $Y^{(0)} \sim h_0$.

Theorem

Let the Markov chain $\{Y^{(r)}\}_{r \geq 1}$ be aperiodic and Harris positive, with $Y^{(0)} \sim h_0$. Moreover let h_r be the marginal probability density of $Y^{(r)}$. Then

$$\lim_{r \to \infty} |h_r(y) - h(y)|_{TV} = 0.$$

Furthermore $|h_r(y) - h(y)|_{TV}$ is decreasing in r.

Ergodic theorem

- The **Ergodic Theorem** is essentially the equivalent of the **law of large numbers** for Markov chains. It is the main justification for using mcmc methods.
- What follows is a slightly simplified version, which is amenable for our purposes.
- Again, the following result holds irrespectively on the initial conditions $Y^{(0)} \sim h_0$.

Theorem

Ergodic Theorem Let the Markov chain $\{Y^{(r)}\}_{r\geq 1}$ be Harris positive with stationary distribution h. Let the function g be integrable w.r.t. to h. Then

$$\frac{1}{R}\sum_{r=1}^{R} g(Y^{(r)}) \to \int g(y)h(y)\mathrm{d}y, \qquad \text{for } r \to \infty,$$

almost surely.

Summary I

- **Sampling** the path of a Markov chain is straightforward from the definition.
- We firstly simulate $Y^{(0)} \sim h_0$. Then we simulate the subsequent values $(Y^{(r+1)} \mid Y^{(r)})$ according to the transition kernel k, assuming it is easy to do so.
- If a Markov chain has a **stationary distribution** h, then simulating from a Markov chain also leads to a practical strategy for simulating from h.
- Because of the previous results, the distribution h_r of $Y^{(r)}$ will eventually **converge** to the stationary law h we wish to simulate.
- Thus, $Y^{(B)}$, for $B > 0$ large enough can be regarded as a sample from h. Moreover, the subsequent values can also be regarded as samples from h, the invariant distribution.

Summary II

- The values $Y^{(1)}, \ldots, Y^{(B)}$ represent the so-called **burn-in period**, namely the values the chain needs to reach convergence.

- The burn-in values should be **discarded**. The choice of B is not always easy in practice

- Hence, the approximations of functionals of interest are based on the values

$$\int g(y)h(y)\mathrm{d}y \approx \frac{1}{R-B} \sum_{r=B+1}^{R} g(Y^{(r)}),$$

  which, once again, we emphasize it relies on the **Ergodic Theorem**.

- What we are still missing are some practical Markov chains algorithms that indeed target a specific stationary distribution.

The Metropolis-Hastings algorithm

Metropolis-Hastings algorithm I

- We are now ready to introduce our first Markov Chain Monte Carlo (MCMC) method: the **Metropolis-Hastings algorithm** (MH).
- This idea goes back to Metropolis et al. (1953) and Hastings (1970).
- Like the acceptance-rejection algorithm, the MH is based on proposing values sampled from an **instrumental proposal distribution**.
- The proposed values are then accepted with a **certain probability** that reflects how likely they are from the target density $h(y)$.
- Under mild conditions, this ensures that the chain will converge to the target density $h(y)$, which is the **stationary distribution**.

Metropolis-Hastings algorithm II

- Set the first value of the chain $y^{(0)}$ to some (reasonable) value.

At the rth value of the chain

i) Let $y = y^{(r)}$ be the current status of the chain. Sample y^* from a proposal distribution

$$q(y^* \mid y).$$

ii) Compute the acceptance probability, defined as

$$\alpha(y^*, y) = \min\left\{1, \frac{h(y^*)q(y \mid y^*)}{h(y)q(y^* \mid y)}\right\} = \min\left\{1, \frac{\tilde{h}(y^*)q(y \mid y^*)}{\tilde{h}(y)q(y^* \mid y)}\right\}$$

iii) With probability $\alpha(y^*, y)$, update the status of the chain and set $y \leftarrow y^*$.

- **We remark** that we do not need to know the normalizing constant K of $h(y) = K\tilde{h}(y)$ because it simplifies in the above ratio.

Detailed balance and reversibility of the MH

- The **transition kernel** of the MH algorithm is therefore the following "mixture"

$$k(y^* \mid y) = \alpha(y^*, y)q(y^* \mid y) + \delta_y(y^*) \int q(s \mid y)[1 - \alpha(s, y)]\mathrm{d}s$$

where $\delta_y(y^*)$ is a point mass at y.

- **Exercise I**. Using the definition of the acceptance probability, verify the following condition:

$$h(y)\alpha(y^*, y)q(y^* \mid y) = h(y^*)\alpha(y, y^*)q(y \mid y^*)$$

- **Exercise II**. From the above equations, conclude that

$$\mathrm{k}(y \mid y^*)h(y) = \mathrm{k}(y^* \mid y)h(y^*)$$

corresponding to the **detailed balance** condition.

- Hence, $h(y)$ is the **stationary law** of a MH process and the chain is reversible.

Detailed balance and reversibility of the MH

- The existence of an invariant stationary distribution is quite a **strong theoretical result**.
- However, one should also check for **irreducibility**, **aperiodicity** and **Harris recurrence** of the MH chain.
- This depends on the proposal distribution $q(y^* \mid y)$ and the stationary density $h(y)$, although it is typically true under very mild conditions.
- Quite general **sufficient conditions** for ergodicity are given in Chapter 7.3.2 of Robert and Casella (2004).
- Failure of MH algorithm typically occurs in presence of a disconnected support for $h(y)$ and/or if the proposal $q(y^* \mid y)$ is not able to explore the support of $h(y)$.

Example: Gaussian distribution

- Suppose we wish to simulate from a Gaussian distribution $N(\mu, \sigma^2)$ using a MH algorithm, whose density is $h(y)$.

- This is obviously a **toy example**, because in practice one would just use rnorm.

- For the proposal distribution $q(y^* \mid y)$, we can use a uniform **random walk**, namely

$$Y^* = y + U, \qquad \text{with} \qquad Y \sim Unif(-\epsilon, \epsilon).$$

  The choice of $\epsilon > 0$ will impact the algorithm, as we shall see.

- Random walks are **symmetric** proposals distributions, so $q(y^* \mid y)$.

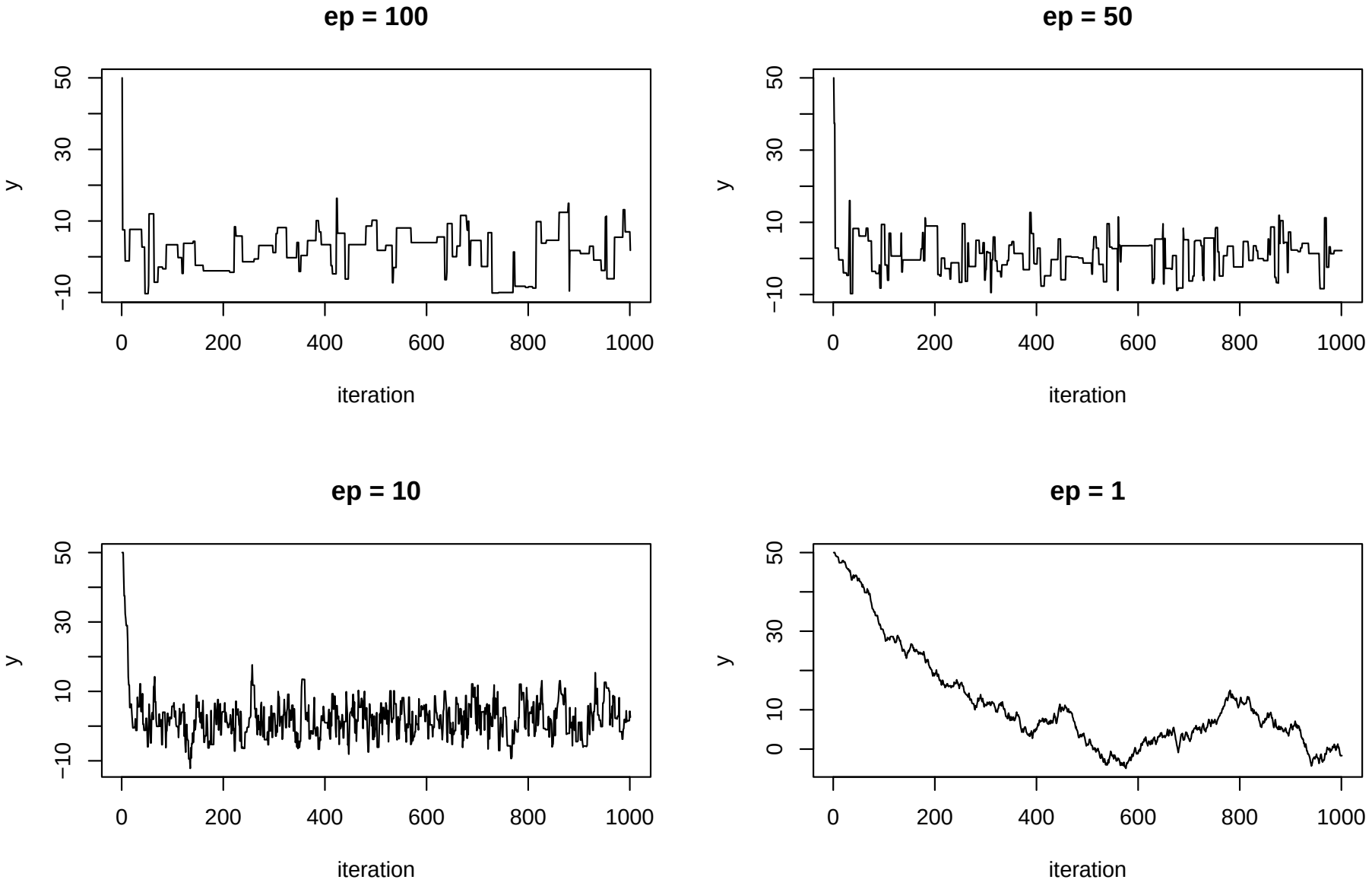
- This means the acceptance probability α is equal to

$$\alpha(y^*, y) = \min\left\{1, \frac{h(y^*)}{h(y)}\right\}$$

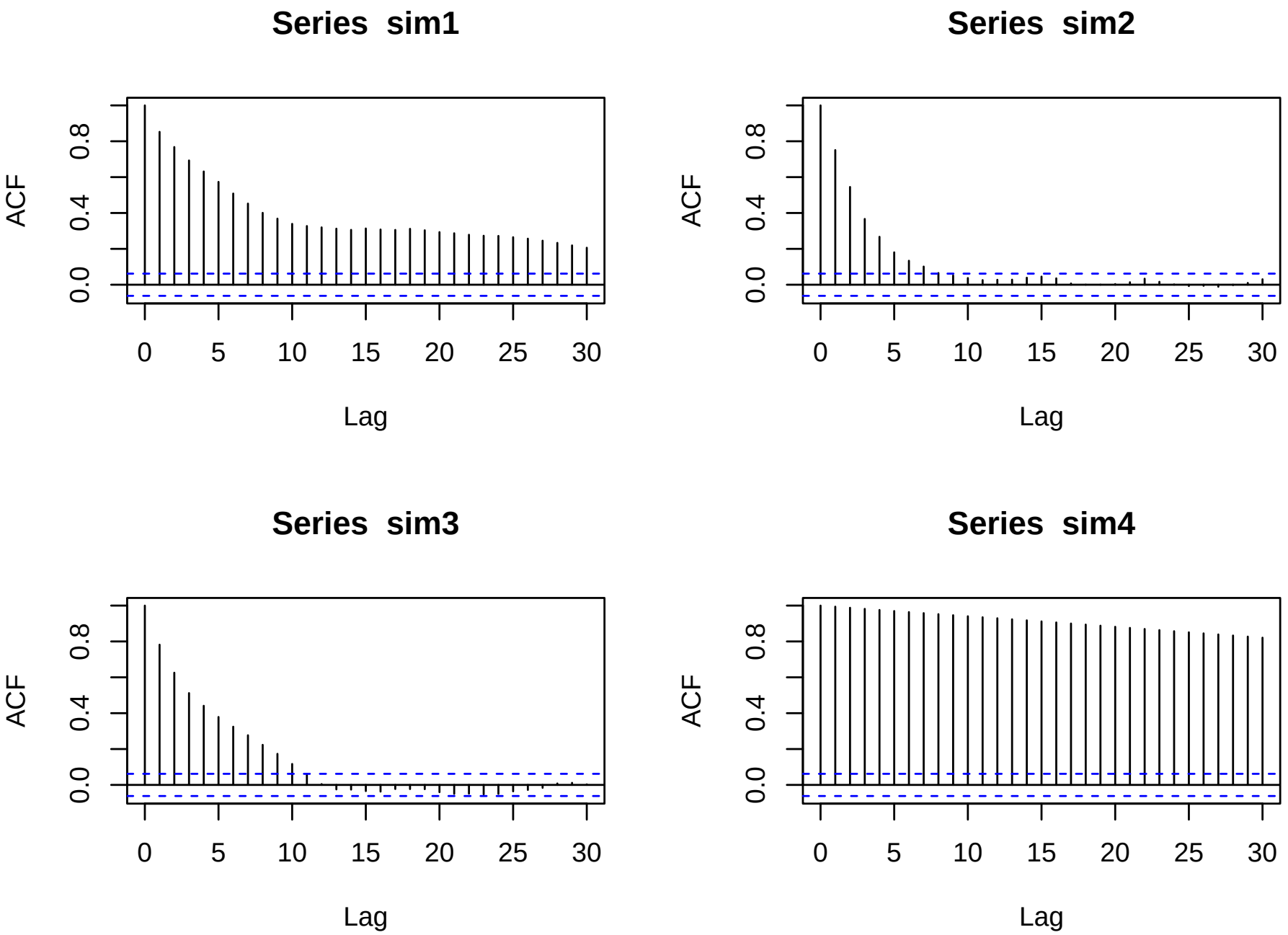
Example: Gaussian distribution

```r
norm_mcmc <- function(R, mu, sig, ep, x0) {
        # Initialization
        out <- numeric(R + 1)
        out[1] <- x0
        # Beginning of the chain
        x <- x0
        # Metropolis algorithm
        for(r in 1:R){
                # Proposed values
                xs <- x + runif(1, -ep, ep)
                # Acceptance probability
                alpha <- min(dnorm(xs, mu, sig) / dnorm(x, mu, sig), 1)
                # Acceptance / rejection step
                accept <- rbinom(1, size = 1, prob = alpha)
                if(accept == 1) {
                        x <- xs
                }
                out[r + 1] <- x
        }
        out
}
```

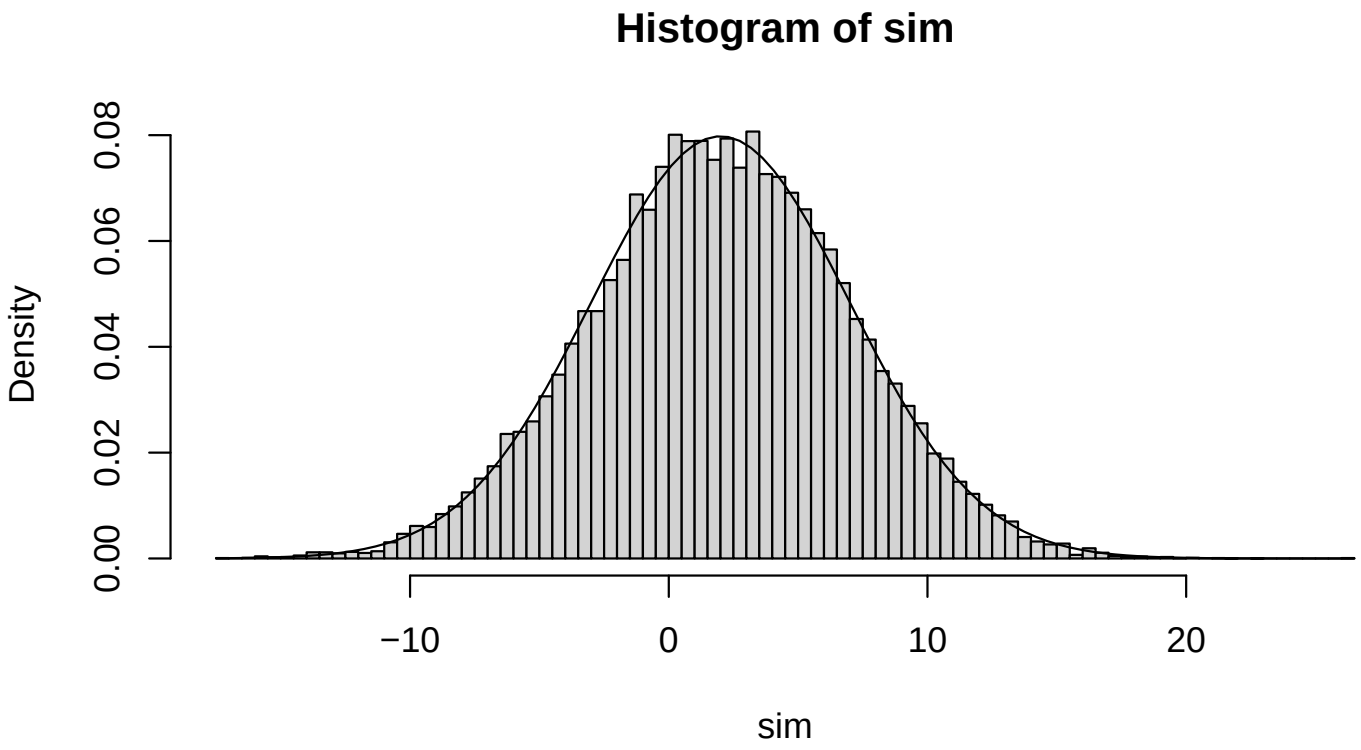
Example: Gaussian distribution



Example: Gaussian distribution



Example: Gaussian distribution




```r
# Simulate the MH chain
sim <- norm_mcmc(50000, mu = 2, sig = 5, ep = 10, x0 = 50)
# Identify a burn-in period
burn_in <- 1:200; sim <- sim[-c(burn_in)]
# Plot the results
hist(sim, breaks = 100, freq = FALSE)
curve(dnorm(x, 2, 5), add = T) # This is usually not known!
```

Hybrid Metropolis-Hastings

- The actual advantage of mcmc over classical sampling methods is actually evident in **high dimensions**. We consider $\boldsymbol{Y}^{(r)} = (Y_1^{(r)}, \ldots, Y_p^{(r)})$.

- An option is to use the "vanilla" Metropolis-Hastings algorithm. However, the proposal distribution is not easy to choose if $p > 2$. Unit B.1 is devoted to this issue.

- An alternative is using a "hybrid" Metropolis-Hastings algorithm. This scheme is also known as **Metropolis-within-Gibbs**.

- The idea is quite simple: iteratively apply a Metropolis-Hastings update to **each coordinate** $Y_j^{(r)}$, according to the proposal distributions $q_j(y_j^* \mid y_j)$.

- Sometimes, updating a block of coordinates rather than univariate components is convenient.

- This algorithms is **ergodic** and has **stationary distribution** $h(y)$, under mild conditions. This should be taken for granted, e.g., Chapter 10.3.3 of Robert and Casella (2004).

Example: bivariate Gaussian

- Suppose we aim at simulating from a bivariate Gaussian distribution whose density is

$$h(y_1, y_2) = \frac{1}{2\pi\sqrt{(1-\rho^2)}} \exp\left\{-\frac{1}{2(1-\rho^2)}(y_1^2 - 2\rho y_1 y_2 - y_2^2)\right\}$$

```
# Density of a bivariate Gaussian (up to a proportionality constant)
dbvnorm <- function(x, rho) {
        exp(-(x[1]^2 - 2 * rho * x[1] * x[2] + x[2]^2) / (2 * (1 - rho^2)))
}
```

- For the proposal distributions $q_j(y_j^* \mid y_j)$, we can again use a uniform random walk, namely

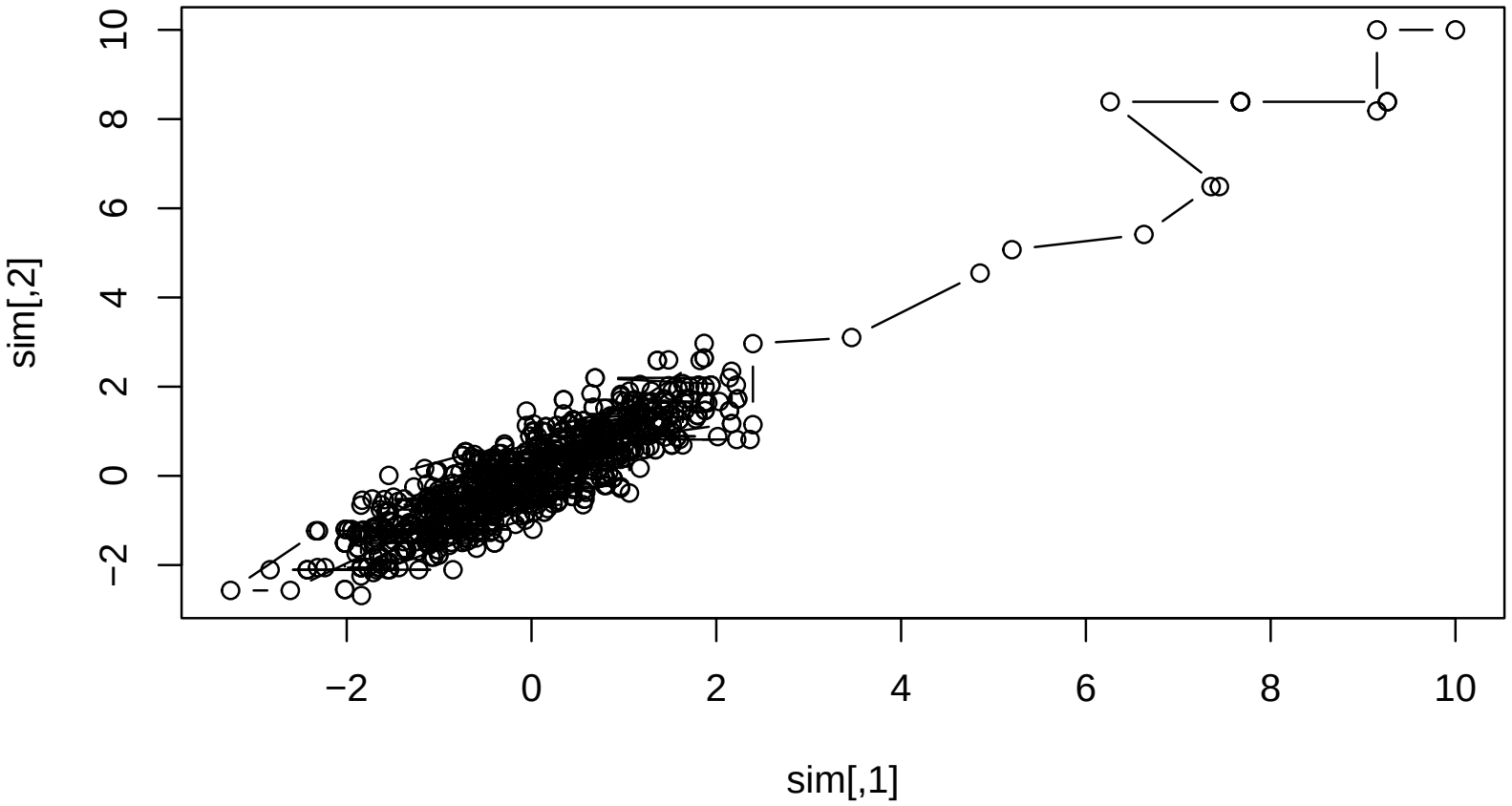
$$y_j^* = y_j + U_j, \qquad U \sim Unif(-\epsilon_j, \epsilon), j = 1, 2.$$

- As before, the choice of j affects the performance of the MH.

Example: bivariate Gaussian

```r
# Hybrid Metropolis (Metropolis-within-Gibbs)
bvnorm_mcmc <- function(R, rho, ep, x0) {
        out <- matrix(0, R + 1, 2)
        out[1, ] <- x0
        x <- x0
        for(r in 1:R){
                for(j in 1:2){
                        xs <- x
                        xs[j] <- x[j] + runif(1, -ep[j], ep[j])
                        # Acceptance probability
                        alpha <- min(dbvnorm(xs, rho) / dbvnorm(x, rho), 1)
                        # Acceptance / rejection step
                        accept <- rbinom(1, size = 1, prob = alpha)
                        if(accept == 1) {
                                x[j] <- xs[j]
                        }
                }
                out[r + 1, ] <- x
        }
        out
}
```

Example: bivariate Gaussian



- Hybrid mh algorithm targeting the stationary density of a bivariate normal with correlation $\rho = 0.8$, with starting point $(10, 10)$.

MCMC with Bayes

Metropolis-Hastings algorithm in Bayesian statistics

- The **Metropolis-Hastings** (MH) algorithm is especially useful for Bayesian inference. In the following, we rephrase the MH using the Bayesian notation.
- Usually, we are interested to sample from the **posterior distribution** of a parameter $\pi(\theta \mid \boldsymbol{X})$.
- Set the first value of the chain $\theta^{(0)}$ to some (reasonable) value.

At the rth value of the chain

i) Let $\theta = \theta^{(r)}$ be the current status of the chain. Sample θ^* from a proposal distribution

$$q(\theta^* \mid \theta).$$

ii) Compute the acceptance probability, defined as

$$\alpha(\theta^*, \theta) = \min\left\{1, \frac{\pi(\theta^* \mid \boldsymbol{X})q(\theta \mid \theta^*)}{\pi(\theta \mid \boldsymbol{X})q(\theta^* \mid \theta)}\right\} = \min\left\{1, \frac{\pi(\theta^*)\mathrm{L}(\boldsymbol{X} \mid \theta^*)q(\theta \mid \theta^*)}{\pi(\theta)\mathrm{L}(\boldsymbol{X} \mid \theta)q(\theta^* \mid \theta)}\right\}$$

iii) With probability $\alpha(y^*, y)$, update the status of the chain and set $y \leftarrow y^*$.

Gibbs sampling

- We now introduce another **Markov Chain Monte Carlo method**: the **Gibbs Sampling**.
- We still wanna sample from the **posterior distribution** $\pi(\boldsymbol{\theta} \mid \boldsymbol{X})$ of $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^p$, given the data.
- Let us **partition** the parameter vector $\theta = (\theta_1, \ldots, \theta_L)$ into L **blocks of parameters**, with $L \leq p$ number of parameters.
- Evenutally, we can have as many blocks as parameters, so that $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_p)$.
- Let $\pi(\theta_\ell \mid -)$ be the so-called **full-conditional** of $\boldsymbol{\theta}_\ell$, that is

$$\pi(\theta_\ell \mid -) = \pi(\theta_\ell \mid \boldsymbol{X}, \theta_1, \ldots, \theta_{\ell-1}, \theta_{\ell+1}, \ldots, \theta_L), \qquad \ell = 1, \ldots, L,$$

  namely the conditional distribution of $\boldsymbol{\theta}_\ell$ given the **data** and the **other parameters**.
- Repeatedly sampling θ_ℓ, for $\ell = 1, ..., L$, from the corresponding full conditionals leads to a mcmc algorithm targeting the posterior distribution $\pi(\theta \mid \boldsymbol{X})$.

Gibbs sampling

- The Gibbs sampler is a special case of **hybrid Metropolis-Hastings**, in which the **full conditionals** are used as **proposal distribution**.

- The general hybrid MG is indeed often called **Metropolis-within-Gibbs**.

- Suppose that $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_p)$. We propose a value updating the jth component, with $\boldsymbol{\theta}^* = (\theta_1, \ldots, \theta_\ell^*, \ldots, \theta_p)$.

- The distribution we want to sample from is the joint posterior

$$\pi(\theta_\ell, \boldsymbol{\theta}_{-\ell} \mid \boldsymbol{X}).$$

- In addition, note that the acceptance probabilities of the hybrid MH algorithm are

$$\alpha_j = \min\left\{1, \frac{\pi(\boldsymbol{\theta}^* \mid \boldsymbol{X}) q(\theta_\ell \mid \theta_\ell^*)}{\pi(\boldsymbol{\theta} \mid \boldsymbol{X}) q(\theta_\ell^* \mid \theta_\ell)}\right\} = \min\left\{1, \frac{\pi(\theta_\ell^*, \boldsymbol{\theta}_{-\ell} \mid \boldsymbol{X}) \pi(\theta_\ell \mid \boldsymbol{X}, \boldsymbol{\theta}_{-\ell})}{\pi(\theta_\ell, \boldsymbol{\theta}_{-\ell} \mid \boldsymbol{X}) \pi(\theta_\ell^* \mid \boldsymbol{X}, \boldsymbol{\theta}_{-\ell})}\right\} = 1.$$

Gibbs sampling

- The **acceptance rate** of the Gibbs sampler is uniformly equal to 1.
- The use of a Gibbs sampler requires the knowledge of the **full-conditional distributions**, from which we should be able to sample.
- The Gibbs sampling is "automatic", in the sense that there are **no tuning parameters** that we need to choose, which is both good and bad news.
- Ergodicity and convergence to the posterior stationary distribution are ensured under **very mild conditions**, i.e. requiring the connectedness of the support.
- The **Hammersley-Clifford theorem** implies that a sufficiently regular joint density can be expressed as a function of the full conditionals.

Example: conditionally-conjugate Gaussian model

- Let us assume the observations $x_1, \ldots, x_n$ are draws from

$$X_i \mid \mu, \sigma^2 \overset{iid}{\sim} N(\mu, \sigma^2)$$

with independent priors $\mu \sim N(m_0, \lambda_0^2)$ and $\sigma^2 \sim IG(a_0, b_0)$.

- The full conditional of the mean μ is

$$\mu \mid - \sim N(m_n, \lambda_n^2), \qquad m_n = \lambda_n^2 \left(\frac{m_0}{\lambda_0^2} + \frac{1}{\sigma^2} \sum_{i=1}^{n} x_i \right), \qquad \lambda_n^2 = \left(\frac{n}{\sigma^2} + \frac{1}{\lambda_0^2} \right)^{-1}.$$

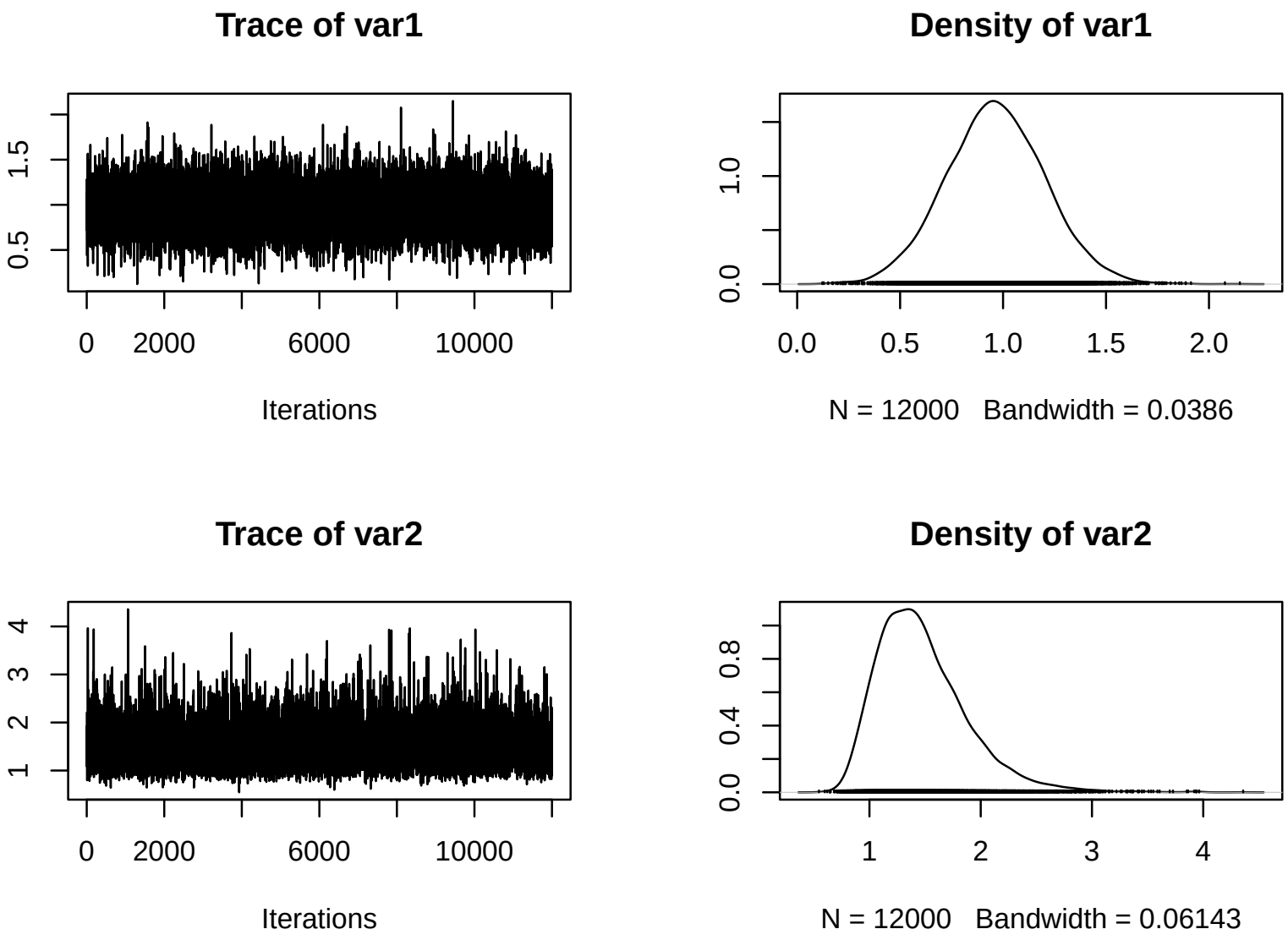
- The full conditional of the variance σ^2 is

$$\sigma^2 \mid - \sim IG(a_n, b_n), \qquad a_n = a_0 + \frac{n}{2}, \qquad b_n = b_0 + \sum_{i=1}^{n} (x_i - \mu)^2.$$

Example: conditionally-conjugate Gaussian model

```r
gibbs_R <- function(x, mu_mu, sigma2_mu, a_sigma, b_sigma, R, burn_in) {
        # Initialization
        n <- length(x); xbar <- mean(x)
        out <- matrix(0, R, 2)
        # Initial values for mu and sigma
        sigma2 <- var(x); mu <- xbar
        for (r in 1:(burn_in + R)) {
                # Sample mu
                sigma2_n <- 1 / (1 / sigma2_mu + n / sigma2)
                mu_n <- sigma2_n * (mu_mu / sigma2_mu + n / sigma2 * xbar)
                mu <- rnorm(1, mu_n, sqrt(sigma2_n))
                # Sample sigma2
                a_n <- a_sigma + 0.5 * n
                b_n <- b_sigma + 0.5 * sum((x - mu)^2)
                sigma2 <- 1 / rgamma(1, a_n, b_n)
                # Store the values after the burn-in period
                if (r > burn_in) {
                        out[r - burn_in, ] <- c(mu, sigma2)
                }
        }
        out
}
```

Example: bivariate Gaussian



Good practice

Implementation of mcmc

- Here we focus on **practical considerations** concerning the implementation with R.
- Higher performance can be achieved using C++ and the Rcpp package (see later).
- This is far from a comprehensive guide about R programming. We will consider a specific model, and we will implement the relevant code in R.

What about BUGS / JAGS / Stan?

- If the performance is not a concern, Stan-like software is a handy tool for practitioners who wish to implement standard Bayesian models.
- Conversely, any non-standard or novel model, i.e., those usually developed by researchers in statistics, may be difficult or even impossible to implement.
- Besides, the "manual" implementation is very useful to gain insights about the model itself and it facilitates a lot the debugging process

Example II: Weibull model for censored data

- We consider an example from survival analysis, i.e., the data are **survival times**, which may be **censored**.
- In this example, we assume that the survival times are iid random variables following a Weibull distribution $Weib(\gamma, \beta)$.
- The observed survival time ti is either **complete** ($d_i = 1$) or **right censored** ($d_i = 0$), meaning that the survival time is higher than the observed t_i.
- The **hazard** and **survival** functions of a Weibull distribution are

$$h(t \mid \gamma, \beta) = \frac{\gamma}{\beta}\left(\frac{t}{\beta}\right)^{\gamma-1}, \qquad S(t \mid, \gamma, \beta) = \exp\left\{-\left(\frac{t}{\beta}\right)^{\gamma}\right\}.$$

- Recall that the **density function** is obtained as $f(t \mid \gamma, \beta) = h(t \mid \gamma, \beta)S(t \mid \gamma, \beta)$.

Likelihood function

- The **likelihood** for this parametric model, under suitable censorship assumptions, is **proportional** to the following quantity

$$\mathrm{L}(\boldsymbol{t}, \boldsymbol{d} \mid \gamma, \beta) \propto \prod_{i=1}^{n} h(t_i \mid \gamma, \beta)^{d_i} S(t_i \mid \gamma, \beta) = \prod_{i:d_i=1} f(t_i \mid \gamma, \beta) \prod_{i:d_i=0} S(t_i \mid \gamma, \beta)$$

  with $\boldsymbol{\theta} = (\gamma, \beta)$ being the parameter vector.

- When performing (Bayesian) inference, note that the likelihood is always defined up to an **irrelevant normalizing constant**, not depending on the parameters $\boldsymbol{\theta}$.

- These irrelevant constants **can and should be omitted** when performing computations, especially if they are expensive to evaluate.

Bad implementation I (use the log-scale)

- In our experiments, we make use the stanford2 dataset of the survival package.
- In the first place, we need to implement the log-likelihood function, say loglik.
- The following implementation of the log-likelihood is correct, but **numerically unstable**.

```
loglik_inaccurate <- function(t, d, gamma, beta) {
        hazard <- prod((gamma / beta * (t / beta)^(gamma - 1))^d)
        survival <- prod(exp(-(t / beta)^gamma))
        log(hazard * survival)
}
# Evaluate the log-likelihood at the point (0.5, 1000)
loglik_inaccurate(t, d, gamma = 0.5, beta = 1000)
# [1] -Inf
```

- The product of several terms close to 0 leads to numerical inaccuracies $\Rightarrow$ **use the log-scale instead**.

Bad implementation II (initialize the output)

- This second coding attempt relies on the log scale and is numerically much more stable than the previous version.
- However, this implementation is inefficient ⇒ **do not increase objects' dimension**.

```
lloglik_inefficient2 <- function(t, d, gamma, beta) {
        n <- length(t) # Sample size
        log_hazards <- NULL
        log_survivals <- NULL
        for (i in 1:n) {
                log_hazards <- c(log_hazards, d[i] * ((gamma - 1) *
                        log(t[i] / beta) + log(gamma / beta)))
                log_survivals <- c(log_survivals, -(t[i] / beta)^gamma)
        }
        sum(log_hazards) + sum(log_survivals)
}
# Evaluate the log-likelihood at the point (0.5, 1000)
loglik_inefficient2(t, d, gamma = 0.5, beta = 1000)
# [1] -873.3299
```

Bad implementation III (avoid for loops)

- This third attempt avoids the previous pitfalls, but it is still quite inefficient ⇒ **use vectorized code whenever possible**.

```
lloglik_inefficient1 <- function(t, d, gamma, beta) {
        n <- length(t) # Sample size
        log_hazards <- numeric(n)
        log_survivals <- numeric(n)
        for (i in 1:n) {
                log_hazards[i] <- d[i] * ((gamma - 1) * log(t[i] / beta) + log(gamma / beta))
                log_survivals[i] <- -(t[i] / beta)^gamma
        }
        sum(log_hazards) + sum(log_survivals)
}
# Evaluate the log-likelihood at the point (0.5, 1000)
loglik_inefficient1(t, d, gamma = 0.5, beta = 1000)
# [1] -873.3299
```

Good implementation

- The following version is both **numerically stable** and **efficient**.

```
loglik <- function(t, d, gamma, beta) {
        log_hazard <- sum(d * ((gamma - 1) * log(t / beta) + log(gamma / beta)))
        log_survival <- sum(-(t / beta)^gamma)
        log_hazard + log_survival
}
# Evaluate the log-likelihood at the point (0.5, 1000)
loglik(t, d, gamma = 0.5, beta = 1000)
# [1] -873.3299
```

- All these versions of `loglik` run in fractions of seconds. However, the `loglik` function must be executed, i.e., $\sim 10^5$ times within a MH algorithm.

- Moreover, several instances of these inefficiencies in more complex models add up.

Benchmarking the code

- To understand which function works better, you need to **test its performance**.
- There exist specialized packages to do so, i.e. R rbenchmark or microbenchmark.
- These packages execute the code several times and report the **average execution time**.
- The column "elapsed" refers to the overall time (in seconds) over 1000 replications.

```r
library(rbenchmark) # Library for performing benchmarking

benchmark(
        loglik1 = loglik(t, d, gamma = 0.5, beta = 1000),
        loglik2 = loglik_inefficient1(t, d, gamma = 0.5, beta = 1000),
        loglik3 = loglik_inefficient2(t, d, gamma = 0.5, beta = 1000),
        columns = c("test", "replications", "elapsed", "relative"),
        replications = 1000
)
#  test            replications        elapsed       relative
#1 loglik1         1000                0.014         1.000
#2 loglik2         1000                0.079         5.643
#3 loglik3         1000                0.412         29.429
```

A matter of style

- **Formatting your code** properly is a healthy programming practice.
- You can refer to https://style.tidyverse.org for a comprehensive overview of good practices in R.
- Quoting the **tidyverse style guide**: "Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread".
- The styler R package automatically restyles your code for you, and it is integrated within RStudio as an add-in.

```
# Good
x <- 5
# Bad
x = 5
```

Reparametrizations I

- When performing (Bayesian) inference, the choice of the **parametrization** strongly impacts computations.
- **General advice**: perform computations on the most convenient parametrization and then transform back the obtained samples.
- As a rule of thumb, you should use parametrizations with **unbounded domains**. This facilitates the choice of proposal distributions and could also improve the **mixing**.
- In our model, the two parameters γ, β are strictly positive. Hence, a common strategy is to consider their logarithm, i.e., $\boldsymbol{\theta} = (\theta_1, \theta_2) = (\log(\gamma)\log(\beta))$.

To log or not to log?

Roberts, G. O. and Rosenthal, J. S. (2009). Examples of adaptive MCMC. Journal of Computational and Graphical Statistics, 18(2), 349–367.

Reparametrizations II

- When reparametrizations are involved, there are two possible modeling strategies. Choose the prior **before** the reparametrization. In our setting, we could let for example

$$\gamma \sim Ga(0.1, 0.1), \qquad \beta \sim Ga(0.1, 0.1).$$

  If you do so, remember to include the **jacobian** of the transformation when considering the transformed posterior!

- Choose the prior **after** the reparametrization. In our setting, we could let for example

$$\theta_1 = \log(\gamma) \sim N(0, 100), \qquad \theta_2 = \log(\beta) \sim N(0, 100).$$

- This strategy is more straightforward as it avoids the extra step of computing the jacobian.

```
logprior <- function(theta) {
        sum(dnorm(theta, 0, sqrt(100), log = TRUE))
}
logpost <- function(t, d, theta) {
        loglik(t, d, exp(theta[1]), exp(theta[2])) + logprior(theta)
}
```

The MH implementation

- Since the space of $\boldsymbol{\theta}$ is unbounded, it is reasonable to select a **Gaussian random walk** as proposal distribution, namely

$$(\boldsymbol{\theta}^* \mid \boldsymbol{\theta}) \sim N_2(\mathbf{0}, 0.25^2 I_2).$$

  The choice of the variance will be discussed in the next slides block.

- Gaussian random walks are symmetric proposals distributions, implying that

$$q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^*) = q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})$$

  which means that their ratio can be simplified $(= 1)$ when computing the acceptance probability α.

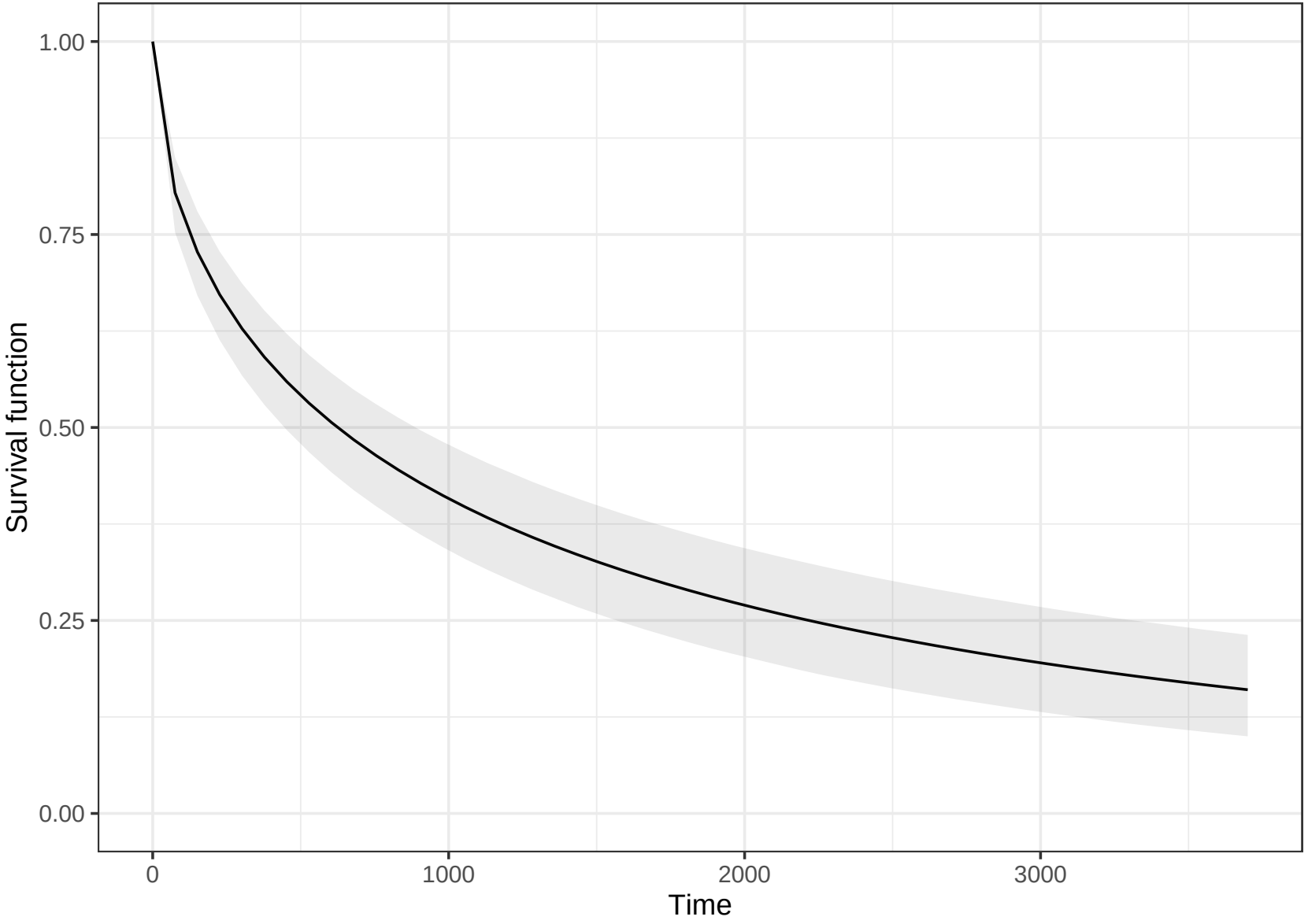
- Compute α using the log scale to avoid numerical instabilities.
- Unfortunately, there is no way to avoid for loops, which are highly inefficient $\Rightarrow$ **This justifies the usage of** Rcpp **and** RcppArmadillo.

Metropolis-Hastings code

```r
RMH <- function(R, burn_in, t, d) {
        out <- matrix(0, R, 2) # Initialize an empty matrix to store the values
        theta <- c(0, 0) # Initial values
        logp <- logpost(t, d, theta) # Log-posterior
        for (r in 1:(burn_in + R)) {
                theta_new <- rnorm(2, mean = theta, sd = 0.25) # Propose a new value
                logp_new <- logpost(t, d, theta_new)
                alpha <- min(1, exp(logp_new - logp))
                if (runif(1) < alpha) {
                        theta <- theta_new; logp <- logp_new # Accept the value
                }
                if (r > burn_in) {
                        out[r - burn_in, ] <- theta # Store the values after burn-in
                }
        }
        out
}
```

```r
# Executing the code
library(tictoc) # Library for "timing" the functions
tic()
fit_MCMC <- RMH(R = 50000, burn_in = 5000, t, d)
toc()
# 0.92 sec elapsed
```

Example: bivariate Gaussian



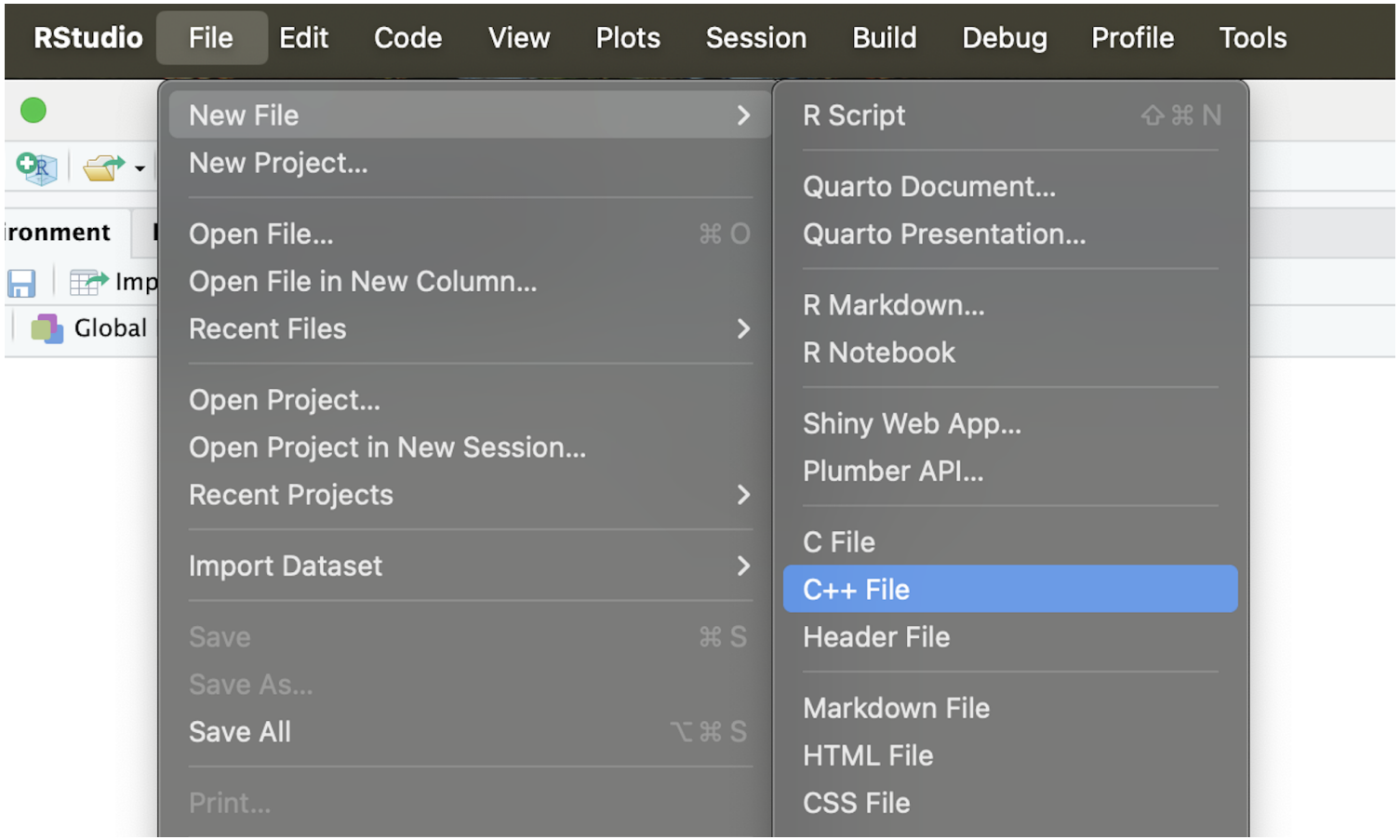
- Posterior mean of the survival function with pointwise 95% credible intervals.

Rcpp & RcppArmadillo

Efficient implementation

- The Rcpp package simplifies the interface between R and C++.
- The package RcppArmadillo extends Rcpp and simplifies the interface between R and armadillo, which is a **"high quality linear algebra library for the C++ language, aiming towards a good balance between speed and ease of use"**.
- The main advantage is that C++ code is usually much faster than R (and python), especially in non-vectorized settings.
- It is tough to be faster than Rcpp, unless your code is written in C++.

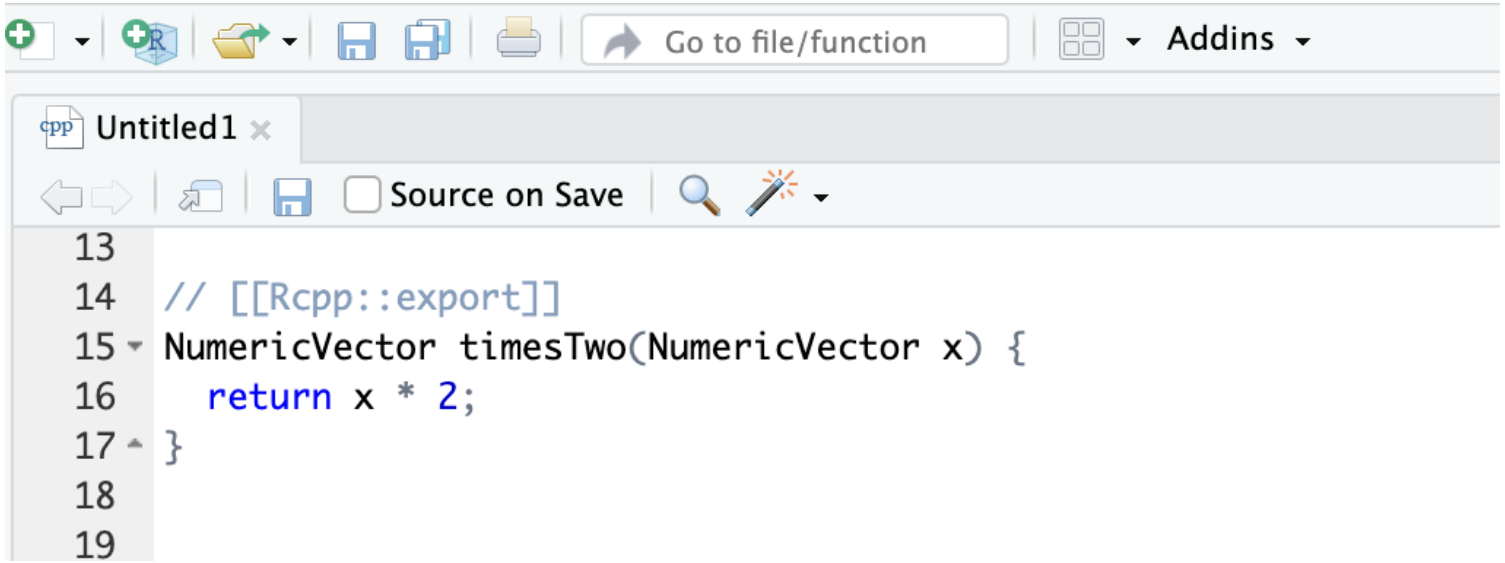
Basic usage



- Nowadays, both packages (Rcpp and RcppArmadillo) are very well integrated within RStudio.

Basic usage

- First, create an empty file, say foo.cpp, containing the C++ code.
- Save the C++ file and **compile it** using the sourceCpp function. Alternatively, you can press the "source" button using RStudio.
- Use the functions contained in the C++ file within R as usual. The functions will appear in the environment.



The sum function in RcppArmadillo

```cpp
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;
using namespace arma;
// [[Rcpp::export]]
double arma_sum(vec x){
        double sum = 0;
        int n = x.n_elem; // Length of the vector x
        for(int i=0; i < n; i++){
                sum += x[i]; // Shorthand for: sum = sum + x[i];
        }
        return(sum);
}
```

```r
sourceCpp("../cpp/sum.cpp")
x <- c(10, 20, 5, 30, 21, 78, pi, exp(7))
arma_sum(x) # sum of the vector x
# [1] 1263.775
sum(x) # sum of the vector x - usual command
# [1] 1263.775
```

Example I: Euclidean distance

- The R code is typically slow in presence of (nested) for loops.
- We are given a matrix X of dimension $n \times p$, whose rows are $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})^\intercal$.
- We are interested in computing the matrix of Euclidean distances D of dimension $n \times n$ whose entries are equal to

$$d_{i,k} = \sqrt{\sum_{j=1}^{p} (x_{ij} - x_{kj})^2}, \qquad i, k \in \{1, \ldots, n\}.$$

```
R_dist <- function(X) {
        n <- nrow(X)
        D <- matrix(0, n, n) # Pre-allocate the output
        for (i in 1:n) {
                for (k in 1:i) {
                        D[i, k] <- D[k, i] <- sqrt(sum((X[i, ] - X[k, ])^2))
                }
        }
        D
}
```

Example I: Euclidean distance

- The corresponding RcppArmadillo o implementation is quite simple as well.

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;
using namespace arma;
// [[Rcpp::export]]
mat arma_dist(const mat& X){
        int n = X.n_rows;
        mat D(n, n, fill::zeros); // Allocate a matrix of dimension n x n
        for (int i = 0; i < n; i++) {
                for(int k = 0; k < i; k++){
                        D(i, k) = sqrt(sum(pow(X.row(i) - X.row(k), 2)));
                        D(k, i) = D(i, k);
                }
        }
        return D;
}
```

Example I: Euclidean distance

- Let us use the USArrests dataset for a quick benchmark.
- The RcppArmadillo implementation is about 150 times faster than the naive R version due to the presence of nested for loops.
- Actually, the RcppArmadillo version is slightly faster the dist built-in R function!

```r
X <- as.matrix(USArrests) # Example dataset
benchmark(
        arma_dist = arma_dist(X), # Armadillo implementation
        R_dist = R_dist(X), # Naive R implementation
        dist = as.matrix(dist(X)), # Built-in R function (C++)
        columns = c("test", "replications", "elapsed", "relative"),
        replications = 1000
)
#  test          replications      elapsed     relative
#1 arma_dist     1000              0.015       1.000
#2 dist          1000              0.080       5.333
#3 R_dist        1000              2.445       163.000
```

Example II: linear models

- R R code is not necessarily slower than Armadillo when linear algebra is involved.
- The RcppArmadillo implementation is about 150 times faster than the naive R version due to the presence of nested for loops.
- Suppose we are interested in obtaining the least squares estimate $\hat{\boldsymbol{\beta}}$ from the design matrix X and the response $\boldsymbol{y}$, namely $\hat{\boldsymbol{\beta}} = (\mathrm{X}\mathrm{X}^\intercal)^{-1}\mathrm{X}^\intercal\boldsymbol{y}$.
- In the first place, let us compare two slightly different R implementations.
- As a rule of thumb, do not invert matrices if the goal is solving linear systems

```r
# Using matrix multiplication commands
lm_coef1 <- function(X, y) {
        solve(t(X) %*% X) %*% t(X) %*% y
}
# Better (no matrix inversion!) and faster implementation
lm_coef2 <- function(X, y) {
        solve(crossprod(X), crossprod(X, y))
}
```

Example II: linear models

- The solve function here can be used directly on the objects X and $\boldsymbol{y}$.

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;
using namespace arma;
// [[Rcpp::export]]
vec lm_coef3(const mat& X, const vec& y) {
        vec coef = solve(X, y);
        return(coef);
}
```

```
set.seed(123)
X <- cbind(1, rnorm(10^4))
y <- rowSums(X) + rnorm(10^4)
cbind(lm_coef1(X, y), lm_coef2(X, y), lm_coef3(X, y)) # Same results
#           [,1]      [,2]      [,3]
# [1,] 0.9909079 0.9909079 0.9909079
# [2,] 1.0060394 1.0060394 1.0060394
```

Example II: linear models

```
benchmark(R_matrix_inv = lm_coef1(X, y),
        R_no_matrix_inv = lm_coef2(X, y),
        Rcpp = lm_coef3(X, y),
        lm = coef(lm(y ~ X, data = cars)),
        columns = c("test", "replications", "elapsed", "relative"),
        replications = 1000
)
#  test              replications         elapsed       relative
# 1 R_matrix_inv             1000           0.365          3.724
# 2 R_no_matrix_inv          1000           0.098          1.000
# 3 Rcpp                     1000           0.141          1.439
# 4 lm                       1000           2.487         25.378
```

- In this case, the RcppArmadillo implementation is approximately as fast as the R version.
- Indeed, the "difficult" part (i.e., solution of the linear system) in all cases is handled by well-optimized C routines.
- The usual lm R functions are slower, but it is calculating many additional quantities.