How do MCMC and Gibbs sampling really work?

Lecture 2 PDF version

Introduction

This short tutorial will guide you through the example of Gibbs sampling shown in class.

As a quick reminder, the example does not really require Gibbs sampling or any form of MCMC to estimate the joint posterior distribution for the parameters. However, because of its specific assumptions it is very helpful because essentially we can determine analytically the full conditional distributions for each parameter (details later). This means that we can directly and repeatedly sample from them (which is a pre-requisite of Gibbs sampling and what `BUGS` does).

This document also includes the R code used to obtain the output.

Set up

As discussed in class, we assume a set up such as the following.

$$
\begin{aligned}
y_i &\overset{iid}{\sim} \text{Normal}(\mu, \sigma^2), \qquad \text{with } i = 1, \dots, n \\
\mu &\sim \text{Normal}(\mu_0, \sigma_0^2) \\
\tau = \frac{1}{\sigma^2} &\sim \text{Gamma}(\alpha_0, \beta_0)
\end{aligned}
$$

Under these assumptions we can *prove* analyticaly that the full conditional distributions for the two parameters μ and τ are

$$
\mu \mid \sigma^2, \boldsymbol{y} \sim \text{Normal}(\mu_1, \sigma_1^2) \qquad \text{with: } \mu_1 = \sigma_1^2 \left(\frac{\mu_0}{\sigma_0^2} + \frac{n\bar{y}}{\sigma^2} \right) \qquad \text{and: } \sigma_1^2 = \left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2} \right)^{-1}
$$

$$
\tau \mid \mu, \boldsymbol{y} \sim \text{Gamma}(\alpha_1, \beta_1) \qquad \text{with: } \alpha_1 = \alpha_0 + \frac{n}{2} \quad \text{and:} \quad \beta_1 = \beta_0 + \frac{1}{2} \sum_{i=1}^{n} (y_i - \mu)^2.
$$

Notice that the full conditionals are **not** the target distributions for Bayesian inference. What we really want is the marginal posterior distributions $p(\mu \mid \boldsymbol{y})$ and $p(\tau \mid \boldsymbol{y})$, which can be simply obtained from the marginal joint posterior $p(\mu, \tau \mid \boldsymbol{y})$.[1]

However, in this case, for each of the two parameters, conditionally on (i.e. given) the other, the posterior is in the same family of the prior — the posterior for μ is still a Normal and the posterior for τ is still a Gamma; what changes is the value of the "hyper-parameters" (i.e. the parameters of these distributions), which move from (μ_0, σ_0^2) to (μ_1, σ_1^2) and from (α_0, β_0) to (α_1, β_1). For this reason, this model is called "semi-conjugated".

These relationships clarify that the (posterior) distribution of the mean μ depends (among other things) on the variance $\sigma^2 = 1/\tau$ as well as that the posterior for the precision τ depends (among other things) on the mean μ. Crucially, ***in this example***, this dependence is known in closed form. A very useful implication of this set up is that it means that we can directly determine not just the distributional form of the posteriors (Normal for the mean and Gamma for the precision), but also the numerical value of the "hyper-parameters" $(\mu_1, \sigma_1^2, \alpha_1, \beta_1)$.

What do we really need to do?

The point of this exercise is to create a routine that can simulate sequentially from these distributions, with the aim of obtaining a sample from the joint posterior distribution $p(\mu, \tau \mid \boldsymbol{y})$. So, once the data y have been observed and we have fixed the values for the parameters of the prior distributions $(\mu_0, \sigma_0^2, \alpha_0, \beta_0)$, we can use, for instance `R`, to simulate from the resulting full conditionals.

At each iteration, we will update sequentially the values of the two parameters: first we update μ by simulating from its full conditional, then we set the value of μ to the one we have just simulated and update the value of τ by randomly drawing from its full conditional. And we repeat this process "until convergence".

Running the example

Data and fixed quantities

Suppose we have observed a sample of $n = 30$ data points, for example, in `R` you may input the data onto your workspace using the following command.

```
# Vector of observed data
y = c(1.2697,7.7637,2.2532,3.4557,4.1776,6.4320,-3.6623,7.7567,5.9032,
7.2671,-2.3447,8.0160,3.5013,2.8495,0.6467,3.2371,5.8573,-3.3749,
4.1507,4.3092,11.7327,2.6174,9.4942,-2.7639,-1.5859,3.6986,2.4544,
-0.3294,0.2329,5.2846)
```

Assume also that you want to set the value of the parameters for the priors as $\mu_0 = 0, \sigma_0^2 = 10000$ and $\alpha_0 = \beta_0 = 0.1$. In R we could define these using the following commands.

```
# "Hyper-parameters" (ie parameters for the prior distributions)
mu_0 = 0
sigma2_0 = 10000
alpha_0 = 0.01
beta_0 = 0.01
```

We may also want to define some "utility" variables, that can be used later on, for instance the sample size and observed sample mean, which we can input in R as in the following code.

```
# Sample size and sample mean of the data
n = length(y)
ybar = mean(y)
```

The actual values of these variables are now stored in the respective "objects" and can be accessed at any point, for instance the commands:

```
n
```

```
## [1] 30
```

```
ybar
```

```
## [1] 3.343347
```

will output the sample size and mean.

Initial values

As seen in the lecture, in order to run the Gibbs sampling algorithm, we need to initialise the Markov chains. This essentially means telling the computer what values should be used at iteration 0 of the process for anything that is a) associated with a probability distribution (i.e. it is not known with absolute certainty); and b) has not been observed.

We can do this in R using the following code.

```
# Sets the "seed" (for reproducibility). With this command, you will
# *always* get the exact same output
set.seed(13)
# Initialises the parameters
mu = tau = numeric()
sigma2 = 1/tau
mu[1] = rnorm(1,0,3)
tau[1] = runif(1,0,3)
sigma2[1] = 1/tau[1]
```

First, we set the "random seed" (in this case to the value 13), which ensures replicability of the results. If you run this code on any machine, you will always and invariably obtain the same results.

Then, we define the parameters mu and tau to be vectors, which in R we do by using the R built-in command `numeric()`. Basically, the command `mu = tau = numeric()` instructs R to expect these two objects to be vectors of (as yet) unspecified length (if you used the command `x = numeric(5)` we would define a vector of length 5).

Finally, we set the first value of `mu` and `tau` to be randomly generated, respectively from a Normal(mean = 0, sd = 3) and a Uniform(0, 3). R has built-in commands to draw (pseudo-)random numbers — typically these are constructed using the prefix `r` (for "random") and a string of text describing the distribution (e.g. `norm` for "Normal", `unif` for "Uniform", `bin` for "Binomial", etc.). The first argument (input) to a call to a `rxxxx(...)` command is the number of values you want to simulate. So, for instance, `rnorm(1000,0,6)` instructs R to simulate 1000 values from a Normal distribution with mean 0 and standard deviation 6 — notice that, unlike BUGS, R parameterises the Normal in terms of mean and sd (instead of the precision). You can check the values that have been selected to initialise your Markov chain by simply typing the name of the variables (or some suitable function thereof).

You can check the values that have been selected to initialise your Markov chain by simply typing the name of the variables (or some suitable function thereof).

```
mu
```

```
## [1] 1.662981
```

```
sqrt(sigma2)
```

```
## [1] 0.9249339
```

Running the Gibbs sampling

Generally speaking, the actual Gibbs sampling is really simple (if the full conditionals are known analytically!) and reduces to code such as the following.

```
# Sets the number of iterations (nsim)
nsim = 1000
# Loops over to sequentially update the parameters
for (i in 2:nsim) {
  # 1. Updates the sd of the full conditional for mu
  sigma_1 = sqrt(1/(1/sigma2_0 + n/sigma2[i-1]))
  # 2. Updates the mean of the full conditional for mu
  mu_1 = (mu_0/sigma2_0 + n*ybar/sigma2[i-1])*sigma_1^2
  # 3. Samples from the updated full conditional for mu
  mu[i] = rnorm(1,mu_1,sigma_1)

  # 4. Updates the 1st parameter of the full conditional for tau
  alpha_1 = alpha_0+n/2
  # 5. Updates the 2nd parameter of the full conditional for tau
  beta_1 = beta_0 + sum((y-mu[i])^2)/2
  # 6. Samples from the updated full conditional for tau
  tau[i] = rgamma(1,alpha_1,beta_1)
  # 7. Re-scales the sampled value on the variance scale
  sigma2[i] = 1/tau[i]
}
```

If you check the code above, you should be able to see that it matches perfectly the mathematical expressions defined for the (updated) parameters of the full posterior distributions. The loop goes from 2 to `nsim` — the first value of the vectors `mu`, `tau` and `sigma2` are filled at initialisation.[2]

Note that the code above produces simulations from the full conditional of the precision τ, which are then rescaled to produce a vector of simulations from the joint posterior distribution for the variance σ^2. It is of course very easy to also rescale these further to obtain a sample of values from the posterior of the standard deviation σ, for example using the following code.

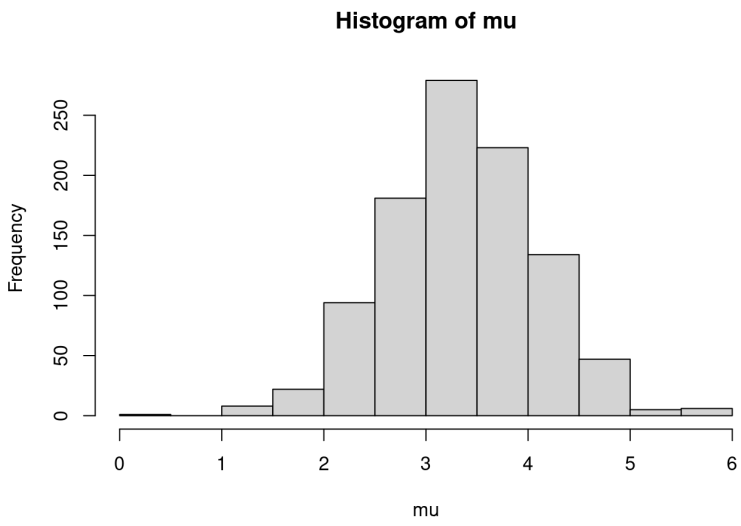
```
sigma = sqrt(sigma2)
```

Once this code has been executed in R your output is made by vectors that, if the procedure has worked (i.e. it has converged), are drawn, at least with a very good degree of approximation, from the joint posterior distribution of the parameters.

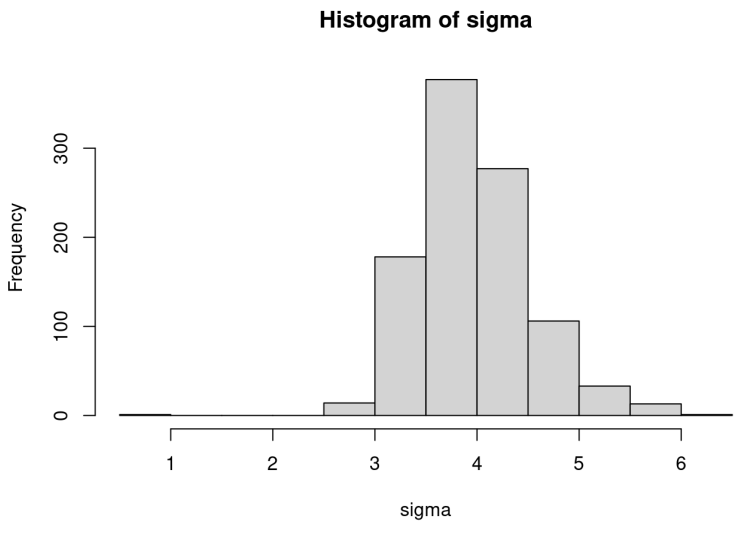
You can also visualise a traceplot — in this particular case, convergence is not an issue (as the model is "semi-conjugated") and even with a single chain, you can get a sense that all the traceplots are "fat, hairy caterpillars", which indicates all is well.

A simple code needed to produce these two plots is the following.

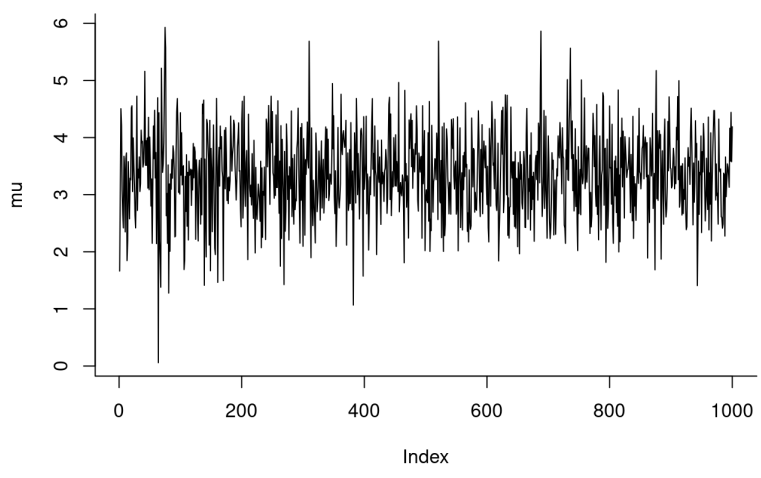
```
# Histograms from the posterior distributions
hist(mu)
```



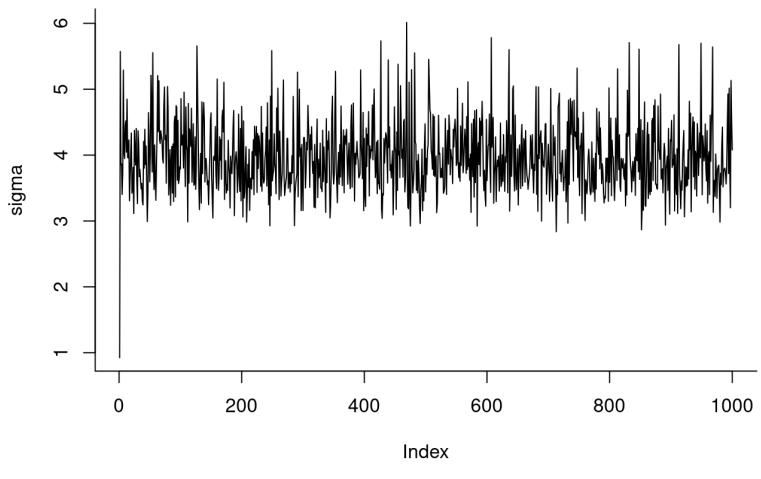

```
hist(sigma)
```




```
# Traceplots
plot(mu,t="l",bty="l")
```

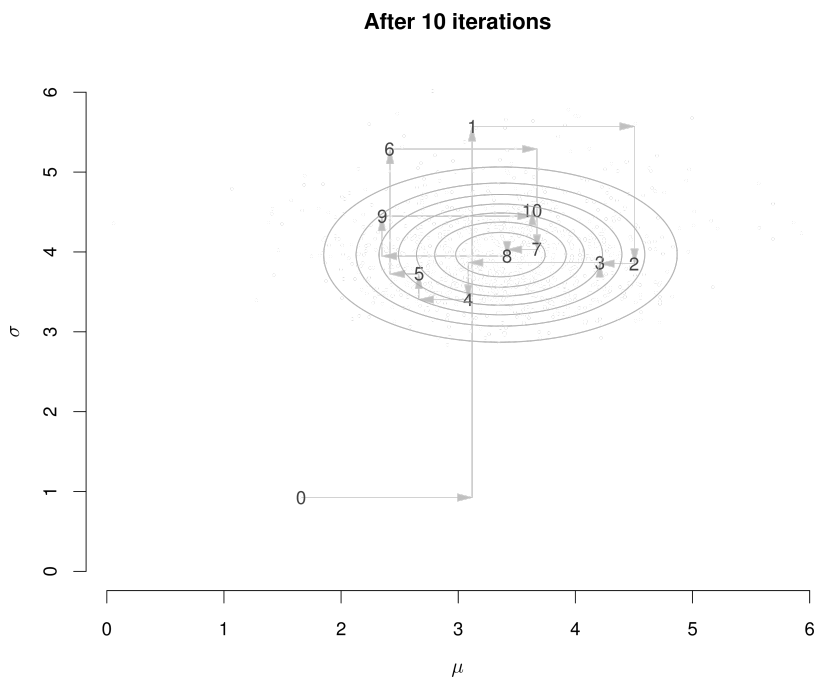



```
plot(sigma,t="l",bty="l")
```



Once again, notice how powerful MCMC is: technically, you may not model directly a (highly non-linear) function of the main parameters; for example, all the computation is made in terms of the precision `tau`, although you may be much more interested in learning the standard deviation sigma. MCMC lets you obtain all the relevant informa- tion on the latter by simply creating simulations from the posterior distributions using the relevant inverse function `sigma = 1/sqrt(tau)`.

We can also depict the trajectories of the MCMC samples in the first 10 iterations of the proecess. The arrows follow the moves of the Markov chain during the updates of the parameters, from the initial



1. The main intuition here is to do with the basic properties of conditional probabilities. Recall that given two events A and B, then by definition $\Pr(A \mid B) = \frac{\Pr(A,B)}{\Pr(B)}$. Now we can extend this to the case of three events A, B and C, by adding C to the conditioning set (i.e. to the right of the "|" symbol throughout the equation) and get $\Pr(A \mid B, C) = \frac{\Pr(A,B|C)}{Pr(B|C)}$. If we replace μ for A, σ^2 for B ,$\boldsymbol{y}$ for C and a probability distribution $p(\cdot)$ for the probability associated with an event $\Pr(\cdot)$, we can write $p(\mu \mid \sigma^2, \boldsymbol{y}) = \frac{p(\mu,\sigma^2|\boldsymbol{y})}{p(\sigma^2|\boldsymbol{y})} \propto p(\mu, \sigma^2 \mid \boldsymbol{y})$. Similarly, $p(\sigma^2 \mid \mu, \boldsymbol{y}) = \frac{p(\mu,\sigma^2|\boldsymbol{y})}{p(\mu|\boldsymbol{y})} \propto p(\mu, \sigma^2 \mid \boldsymbol{y})$. We can then see that each full conditional is proportional to the target distribution (i.e. the joint posterior of all the parameters). Thus, sampling repeatedly from all the full conditionals essentially gives us something that is, broadly speaking, proportional to our target joint posterior distribution. Formal theorems ensure that if we do this long enough, we are guaranteed to actually approximate the target to an arbitrary degree of precision.↩
2. **NB**: there is a slight confusion in the terminology: usually, we refer the initialisation of the process as iteration 0. However, `R` does not let you index a vector with the number 0, i.e. the first element of a vector is indexed by the number 1. Thus, what in the Lecture 3 was indicated as $\mu^{(0)}$ is actually `mu[1]` in the `R` code. Similarly, the updated value for μ at iteration 2 is indicated as $\mu^{(2)}$ in the slides, but as `mu[3]` in the `R` code.↩

PREVIOUS
Covid vaccine: Bayesian modelling