Parallel Slice Sampling

Project for the courses
Bayesian Statistics
Advanced Programming for Scientific Computing

Teresa Pietrabissa
Student ID 782064
teresa.pietrabissa@mail.polimi.it

Simone Rusconi
Student ID 783236
simone1.rusconi@mail.polimi.it

Academic Year 2012-2013
Contents

1 Introduction 1

2 The algorithm 2
   2.1 First step ............................................. 2
   2.2 Second step ......................................... 2

3 Comparison between theory and simulation 4
   3.1 Beta-Bernoulli model in $\mathbb{R}$ ......................... 4
      3.1.1 Exact result ..................................... 4
      3.1.2 Simulated result ................................. 4
   3.2 Normal-Normal model in $\mathbb{R}^3$ (known covariance matrix) 5
      3.2.1 Exact result ..................................... 5
      3.2.2 Covariance matrix without eccentricity .......... 6
      3.2.3 Covariance matrix with strong eccentricity .... 8

4 The speed up 10
   4.1 Beta-Bernoulli model in $\mathbb{R}$ ......................... 10
   4.2 Normal-Normal model in $\mathbb{R}^3$ (known covariance matrix) 11
      4.2.1 Covariance matrix without eccentricity .......... 11
      4.2.2 Covariance matrix with strong eccentricity .... 11
   4.3 Curse of dimensionality .................................. 12

5 Quakes forecasting: our benchmark Bayesian model 14
   5.1 The likelihood distribution ............................... 14
   5.2 The prior distribution ................................... 15
   5.3 A computational problem ................................ 15
      5.3.1 Stick breaking technique .......................... 15
      5.3.2 Squared matrix technique ......................... 16
   5.4 The hyper-rectangle $\mathcal{R}$ ............................ 17
   5.5 A memory problem ...................................... 19
   5.6 The priors hyper-parameters ............................. 19
      5.6.1 The first uninformative prior .................... 20
      5.6.2 The priors updating ................................ 21
   5.7 Unsatisfactory results .................................. 22

6 Our implementation 24
   6.1 Point class ........................................... 24
   6.2 Covariance class ...................................... 24
   6.3 Hyperrectangle class ................................... 25
   6.4 Dataset class .......................................... 25
   6.5 Allocator on global memory struct .................... 26
   6.6 Parallel uniform random number generator struct .... 26
   6.7 Is smaller struct ..................................... 26
   6.8 Measure struct ........................................ 27
   6.9 MCMC class ........................................... 27
      6.9.1 SamplingBeta private function ................. 28
Contents

6.9.2 MakeChain public function .......................... 28

7 Conclusions and future developments 30

8 Acknowledgments 30

A The Code 31
Abstract

To draw a sample of a continuous variable $B$ from a finite measure $g$ with respect to the Lebesgue measure, using a Markov Chain Monte Carlo (MCMC) method, there is an easy algorithm named Slice Sampling. The two main problems of this algorithm are: the solution of an inequality, involving the measure density $g$, which can be hard to find due to the irregularities $g$ can be affected by, and the high dimensionality of the support of the density itself.

Our aim is to create a library, using the Slice Sampling, to draw a MCMC sample from any density $g$ with respect to the Lebesgue measure, in particular to get a realization from the posterior density of a Bayesian model. We will present and discuss a solution and some statistical test applications, using a GPU parallel language, which is, nowadays, becoming more and more commonly employed in the context of Bayesian statistical models.
1 Introduction

MCMC methods provide a general approach for approximating integrals with respect to a wide range of complex distributions.

In a typical Bayesian analysis, the integration measure is the posterior distribution. Let $\beta \in \mathbb{R}^k$ be the argument of the posterior, i.e. the variable of integration, and $X$ the collected data. The posterior distribution $\pi(\beta \mid X)$ usually has a density with respect to the Lebesgue measure (for instance) and is proportional to a measure density

\[ g(\beta) = L(\beta \mid X)\pi(\beta) \]  

(1)

where $L(\beta \mid X)$ is the likelihood of the data, seen as function of $\beta$ and evaluated in the collected data $X$, and $\pi(\beta)$ is the prior density of the model.

Our purpose is to create a library that takes as input a generic measure $g(\beta)$ and returns a MCMC sample from the posterior density

\[ \beta^{(i)} \text{ with } i = 1 \div G \]  

(2)

Notice that it’s not necessary to know the normalizing constant, which is most of the times a computational challenge.

This way, we will able to compute a MCMC sample from many different posterior density distributions.

Finally, as applications, we will compute posterior distributions in some “benchmark” Bayesian models using Parallel Slice Sampling. In particular we will test the goodness of our algorithm on some conjugate models comparing theoretical posterior distribution density with the resulting sample; moreover we will use it to study the well-known dataset “quakes” available in the R software.
2 The algorithm

The algorithm that we are going to follow is the Slice Sampling. This is a simple Gibbs-Sampler, that adds to the variables space the continuous scalar random variable \( U \). The Gibbs-Sampler full conditionals are:

\[
(U | B = \bar{\beta}) \sim U(0; g(\bar{\beta})) \tag{3}
\]

\[
(B | U = u) \sim U(S) \text{ where } S = \{ \beta : g(\bar{\beta}) > u \} \tag{4}
\]

So the algorithm can be summarized as follows.

After initializing \( \beta^{(0)} \) as a suitable value:

1. Sample \( u^{(i)} \) from \( U(0; g(\beta^{(i)})) \).

2. Sample \( \beta^{(i+1)} \) from \( U(S^{(i)}) \), where \( S^{(i)} := \{ \beta \in \mathbb{R}^k | g(\bar{\beta}) > u^{(i)} \} \).

Repeat steps 1. and 2. for \( i = 0 \div G - 1 \) to obtain the sample desired.

MCMC methods are sequential methods as their structure requires it. The same is for the Slice Sampling where, as briefly shown, we get a new realization from the previous one. Indeed, we are sampling a Markov Chain, which is time dependent by definition.

However, each evaluation of the new realization \( \beta^{(i+1)} \) can be split into simple steps. The step 1. is easily computable in a sequential way on CPU, but the step 2. requires the great computing power of GPU architectures.

More details on the two main steps are given in the following subsections.

2.1 First step

The first step requires to sample from \( U(0; g(\beta^{(i)})) \), with \( \beta^{(i)} \) known. This is the easiest step of the algorithm as it consists in sampling from a probability distribution, which is simple to generate from. Moreover, this is a one-dimensional random number generation and, for this reason, it becomes easier to run it with libraries that fully support one-dimensional random number generation itself.

2.2 Second step

The second step consists in sampling \( \beta^{(i+1)} \) from \( U(S^{(i)}) \), which requires resolution of inequality \( g(\bar{\beta}) > u^{(i)} \). Based on the computing power of GPU programming, we followed these simple steps:

1. In a parallel way, sample \( \beta^{(i)}_{(j)} \) from \( U(\mathcal{R}) \), where \( j = 1 \div K, K \) is as big as needed, \( i \) is the fixed main step index, \( \mathcal{R} \subset \mathbb{D} \) is a k-dimensional hyper-rectangle and \( \mathbb{D} \subset \mathbb{R}^k \) is the domain of the function \( g(\bar{\beta}) \).

2. In a parallel way, check if some \( \beta^{(i)}_{(j)} \) make the value of \( g(\beta^{(i)}_{(j)}) \) overcome the value of \( u^{(i)} \).
3. Repeat step 1. and 2. until at least one of the $\beta^{(i)}_{(j)}$ verifies the condition required at previous step.

4. The new realization $\beta^{(i+1)}$ is equal to one of the $\beta^{(i)}_{(j)}$ found at step 2.

The hyper-rectangle $R$ can be obtained in different ways and in general we will ask the user to provide it. Notice that the definition of $R$ is problem dependent: this is why we didn’t implement an algorithm to compute the correct hyper-rectangle through our library, but we will expect the user to provide the most suitable for the specific problem.

Moreover, if we let the user free to set the hyper-rectangle, it can mean that our library could be used to sample from truncated distributions. In fact, if we want to sample $B \sim g$, which support is, for instance, equal to $[0, 1]$, and we set $R = [0, 0.5]$, this is equivalent to sampling from the conditional variable $B \mid B < 0.5$.

Notice that setting $R$ to be wide, means a slowdown of computational time, whereas a hyper-rectangle very small means a lack in variability of the sample. Thus, seems to be better to ask the user to make a preliminary study of the specific problem and to define a proper hyper-rectangle to be given to our library.

Once defined a proper $R$, we can sample from $k$ one-dimensional uniforms, one for each dimension. This way we are able to sample $\beta^{(i)}_{(j)}$ from $U(R)$.

Summarying, we tried to solve the inequality $g(\beta) > u$ with a brute-force attack. In this approach are needed two contrasting features: $K$, the number of attempts, must be very large, but the computational time should remain low. A way to reconcile this two aspects is to take advantage of the great computing power of GPU architectures.
3 Comparison between theory and simulation

The goal of this section is to verify the correctness of the results provided by our library. To do so we tried to get an MCMC sample from conjugate Bayesian models, where the theoretical posterior densities are known. Thus, we compared the output of our simulations with these distributions.

3.1 Beta-Bernoulli model in $\mathbb{R}$

3.1.1 Exact result

We computed an MCMC sample from the Beta-Bernoulli Bayesian model, where the parameter $\beta \in [0, 1]$ is the proportion of success in a Bernoulli trial. As we already know that the prior density

$$\pi(\beta) = \text{Beta}(a, b)$$

is conjugate with respect to the Bernoulli likelihood

$$X_1, ..., X_n | \beta \sim \text{Bernoulli}(\beta)$$

the posterior density generated will be

$$\pi(\beta | X_1, ..., X_n) = \text{Beta}(a + \sum_{i=1}^{n} X_i, b + n - \sum_{i=1}^{n} X_i)$$

Using R software we drew a sample from the Likelihood

$$X_1, ..., X_n | \beta \sim \text{Bernoulli}(\beta)$$

where $n = 200$ and $\beta = 0.3$

obtaining:

$$\sum_{i=1}^{n} X_i = 67$$

The prior density for the model, as already introduced, is the Beta distribution:

$$\beta \sim \pi(\beta) = \text{Beta}(a, b)$$

so that our prior is uninformative with respect to the parameter $\beta$.

So, we computed the exact posterior density distribution, as outlined in equation (7), obtaining:

$$\pi(\beta | X_1, ..., X_n) = \text{Beta}(68.2, 134.2)$$

3.1.2 Simulated result

We compared the previous exact result with the one provided by our library, pretending not to know the normalizing constant and ultimately the exact posterior density distribution.

We set the hyper-rectangle $\mathcal{R} = [0, 1]$, because $\beta \in [0, 1]$.

We put the measure $g$ equal to:
Comparison between theory and simulation

$$\begin{align*}
g(\beta) &= \beta^{67}(1 - \beta)^{133} = L(\beta | X_1, \ldots, X_n) \text{ in (8)} \\
\pi(\beta) &= \beta^{0.2}(1 - \beta)^{0.2} \propto \pi(\beta) \text{ in (10)}
\end{align*}$$

Finally, we got the posterior sample of the model and obtained what was expected from theoretical analysis. As shown in the traceplot in Fig. 1, the posterior sample has a very good behavior and it seems not to have any particular shape. So, it means that our run has reached convergence. If we compare the exact posterior distribution with the histogram of the posterior sample, we find a very good match, which states the goodness of the algorithm (see Fig. 2).

![Traceplot for the posterior sample](image1)

Fig. 1: Traceplot for the posterior sample, obtained from the Beta-Bernoulli model with burn-in=50, thinning=3.

![Comparison between exact distribution](image2)

Fig. 2: Comparison in between exact distribution (in red) and the histogram of the posterior sample.

### 3.2 Normal-Normal model in $\mathbb{R}^3$ (known covariance matrix)

#### 3.2.1 Exact result

Here, we drew an MCMC sample from the Normal-Normal Bayesian model, where we suppose to know the covariance matrix. So, the parameter $\beta \in \mathbb{R}^3$ will be the mean of the Likelihood distribution. We know that the prior density

$$\pi(\beta) = N_3(\eta_0, E_0)$$

(13)
Comparison between theory and simulation

is conjugate with respect to the Normal likelihood

\[ X_1, \ldots, X_n \mid \beta \text{i.i.d.} N_3(\beta, \Sigma) \]  \hspace{1cm} (14)

where \( \Sigma \) is known.

The exact posterior density generated will be

\[ \pi(\beta \mid X_1, \ldots, X_n) = N_3(\eta, E) \]  \hspace{1cm} (15)

where

\[ \eta = (E_0^{-1} + n\Sigma^{-1})^{-1}(E_0^{-1} \eta_0 + n\Sigma^{-1}X) \]  \hspace{1cm} (16)

\[ E = (E_0^{-1} + n\Sigma^{-1})^{-1} \]  \hspace{1cm} (17)

\[ X = \frac{1}{n} \sum_{i=1}^{n} X_i \]  \hspace{1cm} (18)

### 3.2.2 Covariance matrix without eccentricity

Using R software we drew a sample from the Likelihood

\[ X_1, \ldots, X_n \mid \beta \text{i.i.d.} N_3(\beta, \Sigma) \text{ with } n = 200 \]  \hspace{1cm} (19)

where

\[ \beta = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \text{ and } \Sigma = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \]  \hspace{1cm} (20)

So, we are studying the uncorrelated case, or the case without eccentricity of the covariance matrix.

We put the prior hyper-parameters equal to

\[ \eta_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ and } E_0 = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix} \]  \hspace{1cm} (21)

in order to make this distribution uninformative with respect to the parameter \( \beta \).

Computing the exact posterior density, as explained in (15), (16), (17) and (18), we obtained:

\[ \eta = \begin{bmatrix} 1.933714 \\ 1.886313 \\ 2.026890 \end{bmatrix} \text{ and } E = \begin{bmatrix} 0.004997501 & 0 & 0 \\ 0 & 0.004997501 & 0 \\ 0 & 0 & 0.004997501 \end{bmatrix} \]  \hspace{1cm} (22)

Pretending not to know this exact result, we want to obtain a sample from the posterior density distribution, simulating from our library.

We set the hyper-rectangle \( R = [0, 5]^3 \) because the plot of the obtained data \( X_1, \ldots, X_n \) show us that their mean \( \beta \) should be in this set.

We set the measure \( g \) equal to:

\[ g(\beta) = \exp \left\{ -\frac{1}{2} \sum_{i=1}^{n} (\beta - X_i)^T(\beta - X_i) \right\} \exp \left\{ -\frac{1}{20} \beta^T \beta \right\} \]  \hspace{1cm} (23)

\[ \propto L(\beta \mid X_1, \ldots, X_n) \text{ in (19), (20)} \quad \propto \pi(\beta) \text{ in (13), (21)} \]
where $X_i$ are the data drew with (19).

Finally, we got the posterior sample of the model, verifying the theoretical analysis. As shown in the traceplots in Fig. 3, the posterior sample has reached convergence.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{traceplots.png}
\caption{Traceplots for the posterior sample obtained, with burn-in=50 and thinning=10.}
\end{figure}

Computing the mean of the posterior sample $\beta^{(i)}$, with $i = 1 \div 5000$, we obtain:

$$E_i \left[ \beta^{(i)} \right] = \begin{bmatrix} 1.935844 \\ 1.886384 \\ 2.025687 \end{bmatrix} \approx \eta = \begin{bmatrix} 1.933714 \\ 1.886313 \\ 2.026890 \end{bmatrix} \quad (24)$$

which is very close to the exact value computed in (22).

Computing the empirical covariance matrix of the posterior sample $\beta^{(i)}$, with $i = 1 \div 5000$, we obtained:

$$cov_i \left( \beta^{(i)} \right) = \begin{bmatrix} 4.932423e-03 & -6.187030e-05 & -5.417924e-05 \\ -6.187030e-05 & 4.729852e-03 & -8.841787e-06 \\ -5.417924e-05 & -8.841787e-06 & 5.040317e-03 \end{bmatrix} \approx E = \begin{bmatrix} 0.004997501 & 0 & 0 \\ 0 & 0.004997501 & 0 \\ 0 & 0 & 0.004997501 \end{bmatrix} \quad (25)$$

To verify that the MCMC sample $\beta^{(i)}$ is actually drawn from a Normal distribution, in Fig. 4 are plotted the marginal histograms for the sample, showing a gaussian behavior.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{marginal_histograms.png}
\caption{Marginal histograms for the posterior sample obtained.}
\end{figure}
These results show us that the sample drawn gives a good approximation of the exact posterior distribution.

3.2.3 Covariance matrix with strong eccentricity

In order to explore a more difficult situation, we give to the known covariance matrix $\Sigma$ a strong eccentricity.

Using R software we drew a sample from the Likelihood

$$X_1, \ldots, X_n \mid \beta \sim N(\beta, \Sigma) \text{ with } n = 200$$

where

$$\beta = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \quad \text{and} \quad \Sigma = \begin{bmatrix} 1 & 0.95 & 0.9 \\ 0.95 & 1 & 0.95 \\ 0.9 & 0.95 & 1 \end{bmatrix}$$

Thus, the strong eccentricity of this matrix makes the sets $S^{(i)}$ very thin and very hard to find a $\beta$ trial $\in S^{(i)}$.

As before, we put the prior hyper-parameters equal to

$$\eta_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad E_0 = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

in order to make this distribution uninformative with respect to the parameter $\beta$.

Computing the exact posterior density, as explained in (15), (16), (17) and (18), we obtained:

$$\eta = \begin{bmatrix} 2.053353 \\ 2.066150 \\ 2.070536 \end{bmatrix} \quad \text{and} \quad E = \begin{bmatrix} 0.004993228 & 0.004743122 & 0.004493253 \\ 0.004743122 & 0.004992998 & 0.004743122 \\ 0.004493253 & 0.004743122 & 0.004993228 \end{bmatrix}$$

Pretending not to know this result, we got the posterior sample of the model, verifying the theoretical analysis.

We set the hyper-rectangle $R = [0, 5]^3$ because the plot of the obtained data $X_1, \ldots, X_n$ show us that their mean $\beta$ should be in this set.

We put the measure $g$ equal to:

$$g(\beta) = \exp \left\{ -\frac{1}{2} \sum_{i=1}^{n} (\beta - X_i)^T \Sigma^{-1} (\beta - X_i) \right\} \propto L(\beta|X_1, \ldots, X_n) \text{ in (26), (27)}$$

where $\Sigma$ is known (27).

As shown in the traceplots in Fig. 5, the posterior sample has reached convergence. Notice that in this case we get just 500 realizations of the posterior sample, instead of 5000 as the previous case, as the computational time is dramatically higher, due to the sets $S^{(i)}$ thinness.

Anyway, the sample drawn gives a good approximation of the exact distribution.

Computing the mean of the posterior sample $\beta^{(i)}$, with $i = 1 \div 500$, we obtained:

$$\mathbb{E}_i \left[ \beta^{(i)} \right] = \begin{bmatrix} 2.047155 \\ 2.058631 \\ 2.063230 \end{bmatrix} \approx \eta = \begin{bmatrix} 2.053353 \\ 2.066150 \\ 2.070536 \end{bmatrix}$$
Comparison between theory and simulation

**Fig. 5:** Traceplots for the posterior sample obtained, with burn-in=50 and thinning=10.

which is very close to the exact value, computed in (29).

Computing the empirical covariance matrix of the posterior sample $\hat{\beta}^{(i)}$, with $i = 1 \div 500$, we obtained:

$$
cov_i \left( \hat{\beta}^{(i)} \right) = 
\begin{bmatrix}
0.005036919 & 0.004804335 & 0.004487320 \\
0.004804335 & 0.005031871 & 0.004711373 \\
0.004487320 & 0.004711373 & 0.004910644
\end{bmatrix}
\approx
\begin{bmatrix}
0.004993228 & 0.004743122 & 0.004493253 \\
0.004743122 & 0.004992998 & 0.004743122 \\
0.004493253 & 0.004743122 & 0.004993228
\end{bmatrix}
$$

(32)

**Fig. 6:** Marginal histograms for the posterior sample obtained.

To verify that the MCMC sample $\hat{\beta}^{(i)}$ is actually drawn from a Normal distribution, in Fig. 6 are plotted the marginal histograms for the sample, showing a gaussian behavior.

In conclusion, we are satisfied by these results, as good approximations of the theoretical analysis.
4 The speed up

The aim of this section is to show the speed up in computational time through different problems and models.

The comparison is made between two different libraries: the first one is the parallel GPU language library, described so far, whereas the second library, or the comparing term, is a library that follows the same algorithm described before, with the only difference that all the code is implemented in a sequential way.

To make the comparison very challenging for our parallel library, we implement the sequential library in an optimal way.

First of all, in the sequential version, we implemented the second step of the Slice Sampling algorithm following a different procedure, because this is the only parallel step. The others steps of the sequential library follow the same algorithm of the parallel version.

Let $u^{(i)}$ be the output of the first step of Slice Sampling, the second step of the algorithm implies to sample $\beta^{(i+1)}$ from $\mathcal{U}(S^{(i)})$, where $S^{(i)} := \{ \beta \in \mathbb{R}^k | g(\beta) > u^{(i)} \}$.

So, we followed these simple steps:

1. Sample one $\beta_{prop}$ from $\mathcal{U}(\mathcal{R})$, where $\mathcal{R}$ is the hyper-rectangle given by the user (sequential component-wise sampling).
2. Check if this $\beta_{prop}$ make the value of $g(\beta_{prop})$ overcome the value of $u^{(i)}$.
3. Sequentially repeat step 1. and 2. until the $\beta_{prop}$ tested verifies the condition required at previous step.
4. The new realization $\beta^{(i+1)}$ is equal to the first $\beta_{prop}$ that satisfies the test in step 2.

Finally, in order to make the sequential version very performing, we use the Mersenne-Twister generator to sample $\beta_{prop}$ from $\mathcal{U}(\mathcal{R})$. Mersenne-Twister is one of the best random number generator available in the standard C++11.

4.1 Beta-Bernoulli model in $\mathbb{R}$

We compared the time needed by 5 runs to obtain 6000 realizations of the MCMC-sample, with 50 burn-in and 3 thinning, from the Beta-Bernoulli model explained before.

<table>
<thead>
<tr>
<th>SEQUENTIAL RUNS</th>
<th>PARALLEL RUNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.88 s</td>
<td>361.01 s</td>
</tr>
<tr>
<td>0.84 s</td>
<td>360.84 s</td>
</tr>
<tr>
<td>0.87 s</td>
<td>364.88 s</td>
</tr>
<tr>
<td>0.87 s</td>
<td>363.15 s</td>
</tr>
<tr>
<td>0.96 s</td>
<td>363.67 s</td>
</tr>
</tbody>
</table>

In this problem, it’s unadvisable to use a parallel approach.

The problem it’s very easy to be solved in a sequential way, mainly because it is a one
the hyper-rectangle $R$ is very small (i.e. $[0,1]$) and the measure $g$ has a smooth shape (as shown in Fig. 2 and (12)). Also, the parallel code is affected by a loss of time when moving information back and forth from the CPU memory to the GPU memory. But, in order to deal with high-dimensional and more difficult problems the parallel approach became compulsory, as we will see shortly.

4.2 Normal-Normal model in $\mathbb{R}^3$ (known covariance matrix)

4.2.1 Covariance matrix without eccentricity

Here there are the computational time of 5 runs to obtain 1000 realizations of the MCMC-sample, with 50 burn in and 3 thinning, from the Normal-Normal model with known covariance matrix equal to identity matrix (i.e. (20)), as explained in the previous section 3.2.2.

<table>
<thead>
<tr>
<th>SEQUENTIAL RUNS</th>
<th>PARALLEL RUNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>843.48 s</td>
<td>396.54 s</td>
</tr>
<tr>
<td>841.36 s</td>
<td>436.18 s</td>
</tr>
<tr>
<td>776.22 s</td>
<td>512.01 s</td>
</tr>
<tr>
<td>814.61 s</td>
<td>464.23 s</td>
</tr>
<tr>
<td>719.48 s</td>
<td>464.06 s</td>
</tr>
</tbody>
</table>

In this example we perceived a good speed up between the parallel and the sequential library.

4.2.2 Covariance matrix with strong eccentricity

The table shows the computational time of 5 runs to obtain 50 draws of the MCMC posterior sample, with no burn-in and no thinning, from the Normal-Normal model, as in subsection 3.2.3 where the known covariance matrix is set as (27).

<table>
<thead>
<tr>
<th>SEQUENTIAL RUNS</th>
<th>PARALLEL RUNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1601.52 s</td>
<td>63.47 s</td>
</tr>
<tr>
<td>697.2 s</td>
<td>63.50 s</td>
</tr>
<tr>
<td>672.97 s</td>
<td>63.72 s</td>
</tr>
<tr>
<td>2667.27 s</td>
<td>63.63 s</td>
</tr>
<tr>
<td>1281.7 s</td>
<td>63.83 s</td>
</tr>
</tbody>
</table>

This large difference is due to the covariance matrix eccentricity, that makes the sets $S^{[i]}$ very thin.

An interesting thing is that the sequential time is characterized by higher variance with respect to the parallel time, because it is much more affected by the luck of the run than the parallel time. In fact, the parallel version tests a big number of $\beta_{prop}$ at the same time and for this reason it’s less dependent from the luck of the specifical run.
4.3 Curse of dimensionality

We now want to show that the parallel approach could be helpful to deal with curse of dimensionality problems. Thus, the aim is to compare the time needed by the two libraries to obtain the same sample:

\[ X_1, \ldots, X_{100} \overset{i.i.d.}{\sim} N_k(0, I) \]  \hspace{1cm} (33)

where the problem dimension \( k \) starts from 1 and rises up to 17. Thus, we set the hyper-rectangle \( \mathcal{R} \) equal to \([-3.5, 3.5]^k\) and the measure \( g \) equal to \( N_k(0, I) \), without any data.

For each \( k = 1 \div 17 \), we drew 5 times the sample (33), without thinning and burn-in. For each dimension \( k \), the following table shows the mean time, computed on the 5 runs, to obtain (33) from the two libraries.

<table>
<thead>
<tr>
<th>( k )</th>
<th>SEQUENTIAL</th>
<th>PARALLEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000 s</td>
<td>1.406 s</td>
</tr>
<tr>
<td>2</td>
<td>0.000 s</td>
<td>0.562 s</td>
</tr>
<tr>
<td>3</td>
<td>0.002 s</td>
<td>0.586 s</td>
</tr>
<tr>
<td>4</td>
<td>0.002 s</td>
<td>0.596 s</td>
</tr>
<tr>
<td>5</td>
<td>0.006 s</td>
<td>0.628 s</td>
</tr>
<tr>
<td>6</td>
<td>0.018 s</td>
<td>0.646 s</td>
</tr>
<tr>
<td>7</td>
<td>0.038 s</td>
<td>0.674 s</td>
</tr>
<tr>
<td>8</td>
<td>0.130 s</td>
<td>0.682 s</td>
</tr>
<tr>
<td>9</td>
<td>2.402 s</td>
<td>0.738 s</td>
</tr>
<tr>
<td>10</td>
<td>0.654 s</td>
<td>6.257 s</td>
</tr>
<tr>
<td>11</td>
<td>0.574 s</td>
<td>0.852 s</td>
</tr>
<tr>
<td>12</td>
<td>63.588 s</td>
<td>4.936 s</td>
</tr>
<tr>
<td>13</td>
<td>76.146 s</td>
<td>3.052 s</td>
</tr>
<tr>
<td>14</td>
<td>426.608 s</td>
<td>5.842 s</td>
</tr>
<tr>
<td>15</td>
<td>673.860 s</td>
<td>14.974 s</td>
</tr>
<tr>
<td>16</td>
<td>977.562 s</td>
<td>111.704 s</td>
</tr>
<tr>
<td>17</td>
<td>9822.540 s</td>
<td>234.514 s</td>
</tr>
</tbody>
</table>

Table 1: Comparison between parallel and sequential mean time for different problem dimensions.

Notice that for small dimensional problems the sequential approach is better than the parallel approach, but when the problem dimension rises up, the mean time of sequential approach explodes.

The sequential approach also suffers from the problem that the computational time variability is bigger than the parallel one, because the time needed strongly depends on the luck of the specific run, especially for high dimensional problems. So, it’s hard to estimate the time needed to obtain the sample desired using the sequential approach. For each dimension \( k \), the following table shows the standard deviation of the 5 runs computational times.

Thus, the parallel approach it’s compulsory in high dimensional problems, as the one we are going to analize in the following section.
<table>
<thead>
<tr>
<th>$k$</th>
<th>SEQUENTIAL</th>
<th>PARALLEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000 s</td>
<td>2.01 s</td>
</tr>
<tr>
<td>2</td>
<td>0.000 s</td>
<td>4.47e-3 s</td>
</tr>
<tr>
<td>3</td>
<td>4.47e-3 s</td>
<td>8.94e-3 s</td>
</tr>
<tr>
<td>4</td>
<td>4.47e-3 s</td>
<td>5.48e-3 s</td>
</tr>
<tr>
<td>5</td>
<td>5.48e-3 s</td>
<td>4.47e-3 s</td>
</tr>
<tr>
<td>6</td>
<td>1.30e-2 s</td>
<td>5.48e-3 s</td>
</tr>
<tr>
<td>7</td>
<td>1.92e-2 s</td>
<td>5.48e-3 s</td>
</tr>
<tr>
<td>8</td>
<td>9.25e-2 s</td>
<td>1.30e-2 s</td>
</tr>
<tr>
<td>9</td>
<td>4.59 s</td>
<td>3.35e-2 s</td>
</tr>
<tr>
<td>10</td>
<td>2.81e-1 s</td>
<td>6.26 s</td>
</tr>
<tr>
<td>11</td>
<td>4.71e-1 s</td>
<td>1.37e-1 s</td>
</tr>
<tr>
<td>12</td>
<td>9.62e+1 s</td>
<td>5.02 s</td>
</tr>
<tr>
<td>13</td>
<td>1.08e+2 s</td>
<td>9.87e-1 s</td>
</tr>
<tr>
<td>14</td>
<td>6.74e+2 s</td>
<td>4.42 s</td>
</tr>
<tr>
<td>15</td>
<td>1.16e+3 s</td>
<td>4.62 s</td>
</tr>
<tr>
<td>16</td>
<td>8.75e+2 s</td>
<td>1.58e+2 s</td>
</tr>
<tr>
<td>17</td>
<td>1.13e+4 s</td>
<td>6.50e+1 s</td>
</tr>
</tbody>
</table>

**Table 2:** Comparison between parallel and sequential time’s standard deviation for different problem dimensions.
5 Quakes forecasting: our benchmark Bayesian model

A well-known dataset available in the R software is the quakes dataset, that collects 1000 seismic events of magnitude greater than 4.0, occurred in a cube near Fiji since 1964.

For each of them, latitude, longitude and depth of the epicenters and magnitude of the quakes are collected.

The aim of this section is to compute the predictive density distribution for the next earthquake, using the Compositional Method.

To achieve this goal, we need to sample from the posterior density distribution, hence to set out the Bayesian model, fixing prior and likelihood densities.

5.1 The likelihood distribution

Observing the previous data plots (Fig. 7 and Fig. 8), we aren’t able to give data any particular distribution, so we decided to approximate it with a Gaussian Mixture Model. After a first investigation on the dataset, it’s also clear that the magnitude has a smaller variability with respect to the other variables, so we decided to standardize the data to avoid problems linked to heteroscedasticity.

Let $\mathbf{X}_i \in \mathbb{R}^4$, $i = 1 \div 1000$, be the vector representing the standardized latitude, longitude, depth and magnitude of the $i$-th quake; the likelihood density for our data is as follows:

$$X_i \mid (\mathbf{p}_1, \ldots, \mathbf{p}_M, \mathbf{\mu}_1, \ldots, \mathbf{\mu}_M, \mathbf{\Sigma}_1, \ldots, \mathbf{\Sigma}_M) \overset{\text{i.i.d.}}{\sim} \sum_{j=1}^M p_j \mathcal{N}_4(\mathbf{\mu}_j, \mathbf{\Sigma}_j)$$

We decided to keep the number of mixture components $M$ as a fixed parameter. But how many component is our model made by? How to estimate $M$?

To answer these questions, we used the library gmdistribution available in the MATLAB software. This package follows an algorithm, based on maximization of the likelihood, to cluster input data into gaussian clusters.

The input of the algorithm are both the data to be clusterized and the number $M$ in which we want to split the dataset, whereas the output are the knowledge if the algorithm has reached convergence, with the given $M$, and a labels vector to assign each data to its cluster.
Giving $M = 3$ and the standardized data to the `gmdistribution` library, convergence is reached and so we fixed $M$ equal to 3.

The second output of the library, i.e. the labels vector, will be useful to fix the prior hyper-parameters and the hyper-rectangle $\mathcal{R}$, as we will see in subsections 5.4 and 5.6.

Notice that the number of mixture components equal to $M = 3$ means that

$$\beta = (p_1, ..., p_M, \mu_1, ..., \mu_M, \Sigma_1, ..., \Sigma_M) \in \mathbb{R}^{44} \quad (35)$$

Thus, it's not advisable a sequential approach, just only for curse of dimensionality! To have results in reasonable amount of time we resort to the massively parallelization allowed by the GPU architectures.

5.2 The prior distribution

Let be $j, j = 1 \div M$, the index of the $j$ - th cluster, we set the prior distribution for each parameter as follows:

$$(p_1, ..., p_M) \sim \text{Dirichlet}(\alpha_1, ..., \alpha_M) \text{ with } \alpha = \sum_{j=1}^{M} \alpha_j \quad (36)$$

$$\mu_j \sim N_4(\eta_j, E_j) \quad (37)$$

$$\Sigma_j \sim \text{InverseWishart}_4(\Psi_j, \nu_j) \quad (38)$$

where each parameter is independent from the others and so the prior distribution for $\beta$ is the product of these distributions.

5.3 A computational problem

The distributions (36) and (38) imply that:

$$\sum_{j=1}^{M} p_j = 1 \text{ with } p_j \in [0, 1] \forall j = 1 \div M \quad (39)$$

$$\Sigma_j \text{ is a sdp matrix} \quad (40)$$

So we need to guarantee (39) and (40) for each attempt of $\beta$.

5.3.1 Stick breaking technique

To make an attempt of $(p_1, ..., p_M)$, we can sample $(w_1, ..., w_M)$ from $M$ independent $U(0, 1)$, but it's very unlikely that these $(w_1, ..., w_M)$ satisfy (39). A smart solution could be use the stick breaking technique.

Again, we sample $(w_1, ..., w_{M-1})$ from $M - 1$ independent $U(0, 1)$ and then we set:

$$p_1 = w_1 \quad (41)$$

$$p_j = w_j(1 - \sum_{l=1}^{j-1} p_l) \text{ with } j = 2 \div M - 1 \quad (42)$$
\[ p_M = 1 - \sum_{l=1}^{M-1} p_l \]  

(43)

Now, it’s trivial to verify that the \((p_1, \ldots, p_M)\) satisfy (39).

Instead of sampling \((p_1, \ldots, p_M)\), we will sample \((w_1, \ldots, w_{M-1})\) from \(U([0, 1]^{M-1})\) for each \(\beta\) trial.

Before computing \(g(\beta)\), we apply the transformations (41), (42), and (43), in order to give our measure \(g\) the correct parameters.

The output of the library as well will be a selected chain of \((w_1, \ldots, w_{M-1})\): to post-process the results, we need to apply (41), (42), and (43), in order to obtain the true \((p_1, \ldots, p_M)\) values chosen by the algorithm.

5.3.2 Squared matrix technique

We also want for each attempt of \(\Sigma_j\) to be a symmetric definite positive matrix, i.e. (40).

We take advantages from the following theorem:

**Theorem 1.** Let \(A\) be a symmetric matrix, then matrix \(A^* = AA\) is a symmetric definite positive matrix.

**Proof.** \(A^*\) is symmetric because \(A\) is symmetric:

\[ (A^*)^T = (AA)^T = A^T A^T = AA = A^* \]  

(44)

We have to proof that \(A^*\) is definite positive.

\(A\) is symmetric, so its eigenvalues \(\lambda\) are all real and, by definition, they satisfy:

\[ Av = \lambda v \]  

(45)

Right multiplying \(A\) to the previous equation we get:

\[ A(Av) = A(\lambda v) \]  

(46)

\[ (AA)v = \lambda(Av) \]  

(47)

Using the definition of the matrix \(A^*\) and the definition of the eigenvalues \(\lambda\):

\[ A^* v = \lambda(\lambda v) = \lambda^2 v \]  

(48)

This means that the eigenvalues of \(A^*\) are the square of the real eigenvalues of \(A\), so each eigenvalue of \(A^*\) is real and greater than zero.

To make an attempt of \(\Sigma_j\), we fill a symmetric matrix \(W_j\), with uniform random numbers, taken in some proper intervals (i.e. with respect to the hyper-rectangle \(R\)).

Finally, we put

\[ \Sigma_j = W_j W_j \]  

(49)

and the previous theorem guarantees us that \(\Sigma_j\) is a symmetric definite positive matrix.

Again, the coefficients referring to \(\Sigma_j\) of each \(\beta\) trial will be the coefficients of
$W_j$.

These $W_j$ are the trials of the algorithm, so before evaluating $g(\beta)$, we compute the transformation \[ \beta \] in order to give to measure $g$ symmetric and definite positive trials of $\Sigma_j$.

Also the output of our library will be a choosen chain of $W_j$, so, we apply \[ \beta \] in order to obtain the true $\Sigma_j$ chosen by the algorithm.

### 5.4 The hyper-rectangle $\mathcal{R}$

Here we will discuss how we decided to fix the hyper-rectangle $\mathcal{R}$ for the quakes problem. Remembering that we need to use the stick-breaking technique and the squared matrix technique to guarantee \[ \beta \] and \[ \Sigma \], it seems to be clearer to define

\[
\beta = (w_1, \ldots, w_{M-1}, \mu_1, \ldots, \mu_M, W_1, \ldots, W_M)
\]

(50)

If $M = 3$, $\beta$ lives in $\mathbb{R}^{44}$, as told in (35), and the hyper-rectangle $\mathcal{R}$ will be the Cartesian product of 44 intervals.

First of all, the interval referring to each $w_j$, with $j = 1 \div M - 1$, will trivially be equal to $[0, 1]$.

Secondly, we have to fix the intervals for the mean $\mu_j \in \mathbb{R}^4$ of each gaussian mixture component $j = 1 \div M$.

Since each $\mu_j$ lives where data lives, we simply followed this procedure.

Let $v$ be the index of the variables, i.e. $v = 1$ standing for latitude, $v = 2$ for longitude, and so on.

Let $\mu_{jv}$ be the $v-th$ component of the $j-th$ gaussian cluster mean ($j = 1 \div M$).

Let $X_{iv}$ be the $v-th$ component of the $i-th$ standardized data ($i = 1 \div 1000$).

We fix the intervals for each mean as follows:

\[
\mu_{jv} \in [\min_{i=1\div1000} X_{iv}, \max_{i=1\div1000} X_{iv}] \ \forall j = 1 \div M
\]

(51)

Notice that the intervals do not depend on the $j-th$ mixture component, because we don’t want to be informative.

The problem now is how to set the intervals for the coefficients of the matrix $W_j$.

Let’s state some useful key points.

First of all, let the matrix $\hat{\Sigma}_j$ be the empirical covariance matrix of the $j-th$ gaussian mixture component. We can compute it, as we have the labels provided by the gmdistribution library in the MATLAB software. $\hat{\Sigma}_j$ could be a good approximation of $\Sigma_j$.

Secondarily, the spectral decomposition of $\hat{\Sigma}_j$ could be easily computed by R software. If we set $\Lambda_j$ to be the diagonal matrix of eigenvalues from $\hat{\Sigma}_j$, and the columns of matrix $V_j$ to be the corresponding eigenvectors, we can write $\hat{\Sigma}_j = V_j \Lambda_j V_j^{-1}$.

Thirdly, the matrix $W_j$ could be seen as the square root of the matrix $\Sigma_j$ (i.e. (49)), so the matrix $\hat{W}_j = V_j \sqrt{\Lambda_j} V_j^{-1}$ could be a good approximation of $W_j$ and each one of its coefficients could be seen as the “central value” of a given interval.
Then, suppose to know two matrices $L_j$ and $U_j$, such that each coefficient of those matrices will be respectively a lower bound and an upper bound for each coefficient of $\Sigma_j$. It’s intuitive that the lower bound matrix for $W_j$ can be computed as the square root of $L_j$ and the upper bound matrix for $W_j$ can be computed as the square root of $U_j$ (using spectral decomposition).

But, how to estimate $L_j$ and $U_j$?

Our idea is “to modify” $\hat{\Sigma}_j$ until we find proper lower and upper bounds for $\Sigma_j$ coefficients.

In order to have a check of the “modification average” of $\hat{\Sigma}_j$, let’s remember a generalization of Chebyshev’s inequality.

Let $X$ be a $D$-dimensional random vector with expected value $\mu = \mathbb{E}[X]$ and covariance matrix $C = \mathbb{E}[(X - \mu)(X - \mu)^T]$.

If $C$ is a positive-definite matrix, for any real number $t > 0$:

$$\mathbb{P}\left((X - \mu)^T C^{-1} (X - \mu) \leq t\right) > 1 - \frac{D}{t} \quad (52)$$

In our case, let $X = \tilde{X}^j_i$ be the $i$-th standardized data belonging to the $j$-th gaussian mixture component. This implies that $D = 4$ and, for instance, put $t = 10$.

We approximate $C$ with $\hat{\Sigma}_j$ and $\mu$ with $\hat{\mu}_j$, which are, respectively, the empirical covariance matrix and the empirical mean of the $j$-th gaussian mixture component, computable thanks to the labels vector.

The previous equation will be:

$$\mathbb{P}\left((\tilde{X}^j_i - \hat{\mu}_j)^T \hat{\Sigma}_j^{-1} (\tilde{X}^j_i - \hat{\mu}_j) \leq 10\right) > 0.6 \quad (53)$$

We used this equation to estimate $L_j$ and $U_j$.

Let $k \in \mathbb{N}$ be an integer positive number, initially equal to $k = 1$, $j$ be the fixed index of the gaussian mixture component and $i$ be the index of the $i$-th data belonging to the $j$-th cluster.

1. Compute $L^k_j = V_j \left(\Lambda_j/k\right) V_j^{-1}$.

2. Compute $p_j$, the proportion of $\tilde{X}^j_i$ on the index $i$ that make this inequality true:

$$\left(\tilde{X}^j_i - \hat{\mu}_j\right)^T \left(L^k_j\right)^{-1} \left(\tilde{X}^j_i - \hat{\mu}_j\right) \leq 10 \quad (54)$$

3. Repeat 1. and 2., updating $k = k + 1$, until $p_j$ is less than 0.6.

4. Set $\hat{k}$ equal to twice the first $k$ that makes $p_j$ less than 0.6 (to be uninformative and obtain a wider interval).

5. $L_j = V_j \left(\Lambda_j/\hat{k}\right) V_j^{-1}$.

6. $U_j = V_j \left(\hat{k}\Lambda_j\right) V_j^{-1}$.

The idea is to modify the covariance matrix until the Chebyshev’s inequality isn’t satisfied, testing it on the data available. Then, use the matrices that make the inequality false as the bounds for $\Sigma_j$. The bounds for $W_j$ are computed as the square root of $L_j$ and $U_j$, as explained before, using the spectral decomposition.
5.5 A memory problem

The GPU is a very powerful programming unit, but it has a reduced memory space compared to CPU architecture. For example, the computer that we used has a GPU that can store only 150 data on the 1000 available. The solution is to use the same model described above, but run the MCMC with less number of data.

In order to satisfy the capacity bounds, we split the quakes dataset into several homogenous and disjoint sub-datasets, each made by 150 data. These subsets must be representative of the total sample, so the proportion of data among the clusters must be similar to the one in the total dataset and each component of the mixture must have some representative data.

To do so, we used the labels vector provided by MATLAB and we created 6 sub-datasets made by 150 data each and 1 sub-dataset made by 100 data. Let us call A, B, C, D, E, F each sub-dataset made of 150 data, and T the last sub-dataset used as test set.

We want to use all the available information, so we designed an iterative algorithm with specific updating rules.

Firstly, we drew a posterior sample using the A sub-dataset and an uninformative prior.

Secondly, we used the information provided by the newly posterior sample to update the prior for the second run, that will use the B sub-dataset.

Then, we keep on updating the prior of each run with the information provided by the posterior of the previous run, until we use all the 6 sub-datasets of 150 data each. The sixth posterior sample will be our final posterior sample, sure that we used as much data information as we could.

Updating priors, means that we update the hyper-parameters and we keep fixed the functional shape, as in (36), (37) and (38).

Finally, we used the T sub-dataset as a test-set. In order to test the convergence of our iterative method, we computed Log Pseudo Marginal Likelihood (LPML), given by each partial posterior sample, over the T sub-dataset.

![Fig. 9: The figure shows the idea of the iterative method, or the idea to update each prior with the previous posterior.](image)

5.6 The priors hyper-parameters

We have explained the iterative method in the previous section, so now we want to highlight how to update step by step the information provided.

In other words, the purpose of this section is to explain how to fix the prior hyper-parameters of each run.

Remembering that the each run has a prior defined as in (36), (37) and (38), we
fixed the hyper-parameters as follows.

5.6.1 The first uninformative prior

Here we state how to set the first prior hyper-parameters. The prior of the first run must be uninformative.

An easy idea could be to center that prior to expected values, and giving to each distribution a great variability, in order to make it less informative as possible.

To know these values of means and variances, we use all the quakes standardized dataset $X_i$, with $i = 1 \div 1000$, and the labels vector, computed by MATLAB.

In the end, we set these hyper-parameters as follow:

- $\alpha_j$ proportional to the number of data belonging to $j$ – th cluster
- $\alpha$ close to little positive number

This makes the mean of each $p_j$ equal to the proportion of data in $j$ – th cluster and its variance very big.

- $\eta_j$ equal to the sample mean of the $j$ – th cluster
- $E_j$ equal to sample covariance of $j$ – th cluster

This may seem trivial, but it’s a really important point. The reason is that the Slice Sampling suffers from the label switching problem when we compute a Gaussian Mixture Model. To avoid this problem, we have to find a way to indicate to each $\mu_j$ which are the values that can be taken. As gaussian mixture components are stochatically disjoint, we can restrict the sample space for each $\mu_j$ to be the space where corresponding mixture component has not neglectable mass probability.

Fixing the hyper-parameters of $\mu_j$ as written above, we are saying that the prior of the mean of the $j$ – th mixture component “is” the cluster’s data distribution (estimated). We let each $\mu_j$ explore only the space in which its mixture component gives probability mass, avoiding label switching. Notice that we can not affect the prior distribution of $\mu_j$ with more variability, as this would lead to a non-empty intersection between different $\mu_j$ sampling spaces and to the label switching problem.

Moreover, this choice for covariance matrix gives us the advantage of computing an uninformative prior distribution, as data variability is certainly greater then $\mu_j$ variability.

- $\Psi_j$ proportional to sample covariance of $j$ – th cluster
- $\nu_j$ a big integer number

This choice makes the mean of each $\Sigma_j$ equal to the sample covariance of $j$ – th cluster, with a great variability.
5.6.2 The priors updating

Suppose to know the posterior sample \( \hat{\beta}^{(i)} \), with \( i = 1 \div G \), of the current run, how to fix the prior hyper-parameters of the next run?

First of all, since we used the stick breaking technique and the squared matrix technique, remember that

\[
\hat{\beta}^{(i)} = (w_1^{(i)}, \ldots, w_{M-1}^{(i)}, \mu_1^{(i)}, \ldots, \mu_M^{(i)}, W_1^{(i)}, \ldots, W_M^{(i)}) \forall i = 1 \div G
\]

(55)

so, we compute \( \forall i = 1 \div G \) the transformations (41), (42), (43) and (49) in order to obtain

\[
\hat{\beta}^{(i)} = (p_1^{(i)}, \ldots, p_M^{(i)}, \mu_1^{(i)}, \ldots, \mu_M^{(i)}, \Sigma_1^{(i)}, \ldots, \Sigma_M^{(i)}) \forall i = 1 \div G
\]

(56)

and, then, start to work on it, to update the hyper-parameters for next prior.

Remembering (36), (37) and (38), let \( \alpha_j, \alpha, \eta_j, E_j, \Psi_j \) and \( \nu_j \) be the prior hyper-parameters of the next run (\( j \) as usual is the index of the mixture gaussian component). We update those hyper-parameters as follows:

\[
\alpha = \frac{E_i \left[ p_1^{(i)} \right] - E_i \left[ \left( p_1^{(i)} \right)^2 \right]}{E_i \left[ \left( p_1^{(i)} \right)^2 \right] - E_i \left[ p_1^{(i)} \right]^2}
\]

(57)

\[
\alpha_j = \alpha E_i \left[ p_j^{(i)} \right]
\]

(58)


Since \( \eta_j \) and \( E_j \) are respectively the mean and the covariance matrix of the parameter \( \mu_j \), we trivially set:

\[
\eta_j = E_i \left[ \mu_j^{(i)} \right]
\]

(59)

\[
E_j = cov_i \left[ \mu_j^{(i)} \right]
\]

(60)

Then, to be uninformative we set:

\[
\nu_j = 6 \ \forall j = 1 \div M
\]

(61)

Finally, since the expected value of \( \Sigma_j \sim \text{InverseWishart}_4(\Psi_j, \nu_j) \) is equal to \( \frac{\Psi_j}{\nu_j-5} \), we set:

\[
\Psi_j = (\nu_j - 5) E_i \left[ \Sigma_j^{(i)} \right]
\]

(62)

In conclusion, we are ready to run our iterative method.
5.7 Unsatisfactory results

Since we used the stick breaking technique and the squared matrix technique, remember that each run provides:

$$\beta^{(i)} = (w_1^{(i)}, \ldots, w_{M-1}^{(i)}, \var_1^{(i)}, \ldots, \var_M^{(i)}, W_1^{(i)}, \ldots, W_M^{(i)}) \forall i = 1 \div G$$

(63)

so, we compute $\forall i = 1 \div G$ the transformations (41), (42), (43) and (49) in order to obtain

$$\beta^{(i)} = (p_1^{(i)}, \ldots, p_M^{(i)}, \var_1^{(i)}, \ldots, \var_M^{(i)}, \Sigma_1^{(i)}, \ldots, \Sigma_M^{(i)}) \forall i = 1 \div G$$

(64)

which are the posterior parameters.

We ran each posterior chain, obtaining 2000 realizations of the MCMC-sample, with 50 burn-in and no thinning.

All the traceplots obtained have a good behavior, they seem not to have any particular shape. So, it means that our runs have reached convergence. For example, we report the weights $p_j^{(i)}$ traceplots from the last chain.

Fig. 10: Traceplots for the weights $p_1$, $p_2$ and $p_3$ of last posterior chain, with burn-in=50 and no thinning.

It becomes clear the presence of a problem if we take a look at the LPML, computed step by step.

Fig. 11: Log Pseudo Marginal Likelihood, given by each run, computed on the test-set.

It’s clear that we didn’t obtain the expected increasing behavior, such that the LPML reaches an asymptote, which would have meant reaching the convergence. It seems rather that LPML evolves around random values.
We decided to compute the predictive sample anyway, using the Compositional Method on the last sampled chain. Of course, we reversed the standardization because the predictive sample obtained is standardized, as the data are. We obtained these approximations of the marginal predictive distributions:

The expected multimodality of these distributions it’s not evident, which is a clear sign that something went wrong. Moreover, if we compare the data with the predictive sample, it’s clearer that the forecast is uninformative.

The principal reason for these unsatisfactory results is the loss of information caused by each step. Our problem is that the Gaussian Mixture Model needs a large amount of data to provide good results. Besides, we only give partial information of the dataset to each run.

In conclusion, this loss information will be the guideline for our future developments.
6 Our implementation

In this section we would like to highlight the content of our parallel code. We used the library thrust in order to interface between two programming languages: CUDA and C++.

6.1 Point class

The point class shown in Fig. 14 handles vector $\beta$ as an array of float of template dimension D. This choice is due to the problems using CUDA programming: we can’t allocate memory space which is not known at compile time or which is dynamically allocated, as CUDA is a C based language. Hence we decided to implement a class to handle vector $\beta$ as a static array. In particular we implemented some useful functions to access and modify $\beta$ in the rest of the code:

- **set_coord, get_coord and operator[]**: to set (and read) the value of the i-th component of an object of type point
- **dimension**: to get the dimension of the object of type point, as the dimension is not known a priori (the class is a template class over the dimension parameter D)
- **dot**: to implement the dot product between two objects of type point
- **difference**: to implement the difference between two objects of type point

6.2 Covariance class

The covariance class Fig. 16 is to deal with small dimension generic squared matrices. Our purpose is to give the user a tool to handle squared matrices, but in the examples shown in this paper, we only used it with symmetric matrices. Using the transformation in (49), we stored the covariance matrices in an object of type covariance.

As for the point class, we implemented the class with just one member of type array of template dimension $D \times D$.

Here are the most important functions that we implemented for this class.

- **operator()**: to read and write on a matrix in the i-th and j-th position
- **operator =**: to copy a matrix into another object of class type covariance
- **sum**: to perform the sum of two matrices
- **product**: there are three different types of product that we implemented, as in our code we need to perform the product with a constant number, with an object of class type point, or with another matrix
Our implementation

- **trace** and **determinant**: to perform trace and determinant of a matrix (note that to compute determinant we used the minor technique)

- **identity**: to initialize a generic identity matrix of template dimension $D \times D$

- **inverse**: to compute the inverse of a matrix

### 6.3 Hyperrectangle class

The hyper-rectangle class Fig. [17] deals with the support of the measure $g$, which is the support where we look for the new vector $\beta$ for our Markov Chain. As the support is a hyper-rectangle, this class has two private members which are arrays of same template length $D$, one for the minimum and one for the maximum values of the bound over a component of vector $\beta$. To implement functions in the class, we used the library thrust object type thrust::pair, which has the same structure as the standard library object std::pair, but in this case we are able to use the pair object onto the GPU memory. This is crucial, as the parallel step of our algorithm needs to produce random numbers over the support $R$. Here are the functions implemented for the class.

- **set_bound**, **get_bound** and **operator[]**: to write and read, respectively, the i-th component bound. This function returns a thrust::pair object, so that we get to read bounds also when we are working in the GPU memory

- **dimension**: in order to know any time what is the dimension of the hyper-rectangle, as the class is a template class over the dimension parameter

### 6.4 Dataset class

Dataset class Fig. [18] is to handle the dataset of our statistical analysis. Infact, we need to load onto both the CPU and GPU memory the data $X$, in order to evaluate the generic measure $g$, which can be a function of $X$.

As the dataset of a generic statistical analysis is most of the times characterized by high dimensionality, to load as many data as possible onto the GPU memory, we decided to use float type instead of double type data.

Dataset is a matrix type stored as an array of template dimension $R \times C$, where $R$ is the template parameter for number of rows, while $C$ is for the number of columns of the dataset.

- **nrow** and **ncol**: to know number of rows and columns, as the class is template over this two parameters

- **operator()**: to read and write the i-th and j-th dataset position, where matrix is filled rowwise
• **operator=:**
  to copy a dataset object type into another dataset object

• **diff_point_row:**
  to perform difference between a row of the dataset and a point type object, as
  we need in our algorithm to compute \(X[i] - \beta\) when we evaluate measure \(g\)

6.5 Allocated on global memory struct

This struct Fig. 19 is to load objects on the GPU memory from the CPU memory. In
particular, to load the hyper-rectangle we used a point type object and implemented
a function `allocate_device_rect` that saves each bound for different components of
\(\beta\) one after the other. Regarding vector \(\beta\), on the GPU we compute as many vectors
as the number of threads working. In order to save all these in the same container, we
used the thrust object thrust::device_vector, which is a dynamically allocated object
working specifically onto the GPU memory space. Each element of this vector is set to
be a point object type, so that each element of the thrust::device_vector is a \(\beta\) sampled
onto the GPU memory by one of the threads. The function `allocate_beta_sampled`
is to reserve memory, onto the GPU memory space, for the thrust::device_vector.

6.6 Parallel uniform random number generator struct

The struct that we now present is a struct that uses public inheritance of basic class
thrust::unary_function. This way our struct is an adaptable unary function. The
parallel uniform random number generator struct Fig. 20 performs the sampling step
of a \(\beta\) trial. The constructor of the struct loads the hyper-rectangle \(\mathcal{R}\) and the seed,
together with other parameters, for the random engine. The `operator()` takes an
unsigned_int object type as thread’s id and returns a point\(<D> object type as the \(\beta\)
trial for the corresponding thread.

In particular, to generate a new value we used the thrust::random::minstdrand
generator, a version of the Minimal Standard random number generation algorithm,
and the thrust::uniform_real_distribution function to sample uniformly in between
\((0, 1)\). Note that to sample over the correct interval \((a, b)\), given by bounds stored in
the hyper-rectangle, we then shift the generated random number according to the
correct bound. As this operation is done by the GPU threads, this is the reason why
we need to have a point object type for the hyper-rectangle when working on GPU
memory (see subsection 6.5). In the end notice that this struct too is a template
struct over the dimension parameter \(D\) of the vector \(\beta\).

6.7 Is smaller struct

The struct is smaller Fig. 21 deals with the problem of evaluating measure \(g\) for each
\(\beta\) vector trial generated (see subsection 6.6) and testing the inequality \(g(\beta) > u\),
where \(u\) is the current value for the ancillary variable \(U\). The struct has a constructor,
in order to load measure \(g\) and value \(u\) onto the GPU memory, and `operator()`
to test the condition, given a vector \(\beta\) from the thrust::device_vector object (see
subsection 6.5), returning a boolean value (true if the inequality is not satisfied,
false otherwise). The struct is a template struct over two parameters: $M$, being the template name of measure $g$, and $D$, the dimension of the generic vector $\beta$.

### 6.8 Measure struct

This is a struct that each user will need to implement everytime for a new measure $g$. This is a choice that we made in order to give the final user more freedom in implementing its own measure, as a general function or either the product of likelihood and prior densities. The measure class must have a constructor and the `operator()` method. These two functions need to carry both _host_ and _device_ attributes, to have a copy of them into the host compilation flow and into the device compilation flow. `operator()` must take a point $<D>$ object type and return a double object type. “We offer the users endless ways to shoot themselves in the feet”.

In the example shown in Fig. 15 is implemented the measure $g$ for the quakes dataset (see section 5). The struct we are analyzing has multiple members: the dataset $X$, to handle the dataset of the earthquakes, and all the prior’s hyper-parameters for the model. It has a default constructor and `operator()` to evaluate $g(\beta)$. This operator is the heart of the struct, it holds the implementation of the function characterizing the measure we want to work with.

In evaluating $g(\beta)$, we decided to cast to double everything which is possible, as the measure $g$ most of the times returns very small numbers ($\sim 0$). This way we are sure to have double precision operations.

### 6.9 MCMC class

This is the class where all the algorithm is processed. A brief look at the public members (Fig. 22): there are two constructors, one setting stochastic seed, the other setting also the hyper-rectangle and the first $\beta$, i.e. $\beta^{(0)}$, such that each component is the middle point of corresponding bound; there is a default destructor and two setters, one for the hyper-rectangle and one for $\beta^{(0)}$, to be used together with the first constructor; there is function `SaveChain`, to save sampled chain to a generic std::ostream; finally function `MakeChain`, which calls private functions to build the chain (see subsection 6.9.2).

Regarding private members (Fig. 22), there is a member for each main actor of our algorithm:

- **my_measure**:  
  a template $M$ object type for generic measure

- **R**:  
  a hyper-rectangle object type, template over the dimension parameter $D$, to store the hyper-rectangle

- **u**:  
  to store current value of ancillary variable $U$

- **beta**:  
  a point $<D>$ object type, for current $\beta$ vector
• **allocator on global:**
an allocator on global type object type, to store memory for hyper-rectangle \( R \) and thrust::device_vector **beta_sampled** onto GPU memory

• **num_threads:**
to define the number of threads to be used for the parallel step

• **m_seed:**
to save the stochastic seed for each run

• **counter_u** and **counter_beta:**
parameters for the engine generator to discard already sampled numbers

There are two void private functions, **samplingBeta** (see subsection 6.9.1), and **samplingU**, to sample a value \( u \) from a uniform density distribution over the support \((0, g(\beta))\) using thrust::random::minstd_rand engine and thrust::uniform_real_distribution function.

Notice that both the copy constructor and the copy assignment are private members of the class, so this way this is a non-clonable class, which is something compulsory as the class is characterized by a complicated structure and huge amount of memory space needed for all members.

6.9.1 **SamplingBeta** private function

This is the sampling step where we need to introduce parallel programming in order to speed up our algorithm. As explained in section 2.2 we first need to sample as many \( \beta \) proposals as the number of threads involved in the algorithm, where each component of each \( \beta \) is uniformly sampled over the support defined by the corresponding bound in the hyper-rectangle \( R \). To do so we used the thrust function **thrust::transform**. It applies the call operator of the functor **parallel_uniform_random** to each element of a threads’ ids thrust vector (see subsection 6.6). It stores the outputs into the thrust::device_vector **beta_sampled**.

Secondarily to this first step, we need to test the inequality \( g(\beta) > u \), in order to select those \( \beta \) trials that actually satisfy the condition. To do so we used the thrust function **thrust::remove_if**: we remove from the thrust::device_vector **beta_sampled** all elements for which the call operator of class is smaller (see subsection 6.7) returns a true value. We repeat these two steps until we find at least one element of **beta_sampled** satisfying the condition. We set our new realization **beta** to be equal to the first element of the device_vector obtained.

6.9.2 **MakeChain** public function

This public function (see Fig. 24) is the one responsible of calling private functions **samplingU** and **samplingBeta** in order to sample the entire Monte Carlo Markov Chain. First of all we initialize all members which are not initialized through the constructor function (i.e. **num_threads**, **counter_u**, **counter_beta** and **m_seed**), we copy the measure \( g \) reference to our member **my_measure**, we load onto GPU memory, through **allocator on global** member, the hyper-rectangle \( R \), a member of MCMC class built through the constructor, and we initialize an std::vector<point<D>> to be our MCMC chain sampled. Notice that this function
returns the final built chain. 
There are three parameters arguments to be specified when calling MakeChain function, besides measure \( g \): \( G \), length of final chain, \textbf{burn}, number of \( \beta \) sampled to be discard at the very beginning of building process of our chain (the burn-in part of the chain), \textbf{tin}, specifying the number of \( \beta \) sampled to be discard after saving one (thinning process to reduce correlation between two steps of the chain). What we do is that we call iteratively private functions \texttt{samplingU} and \texttt{samplingBeta} one after the other, and according to parameters \textbf{burn} and \textbf{tin} we build up MCMC chain to be returned.
7 Conclusions and future developments

The power of Slice Sampling is that it’s a generic algorithm and a big class of density distributions is suitable to be taken as input of our library. Unfortunately, efficiency and generality are contrasting features, so the computational time is a Slice Sampling problem. Using GPU architectures, we try to solve the problem, sometimes obtaining good results.

The Gaussian Mixture Model is very flexible and it allows us to describe unknown distributions, as the quakes one.

GMM drawback is that it needs a large amount of data, so it’s in contrast with the GPU features.

Thus, a possible future development could be to find a way to load all available data onto the GPU memory. It could mean to use a better technique to store data on our programming unit, or it could mean to use a better architecture, characterized by a greater memory space.

8 Acknowledgments

We wish to express our thanks to Professor Alessandra Guglielmi, Professor Luca Formaggia, PHD Researcher Mattia Penati and Post-Doc Researcher Francesca Ieva. They have been lighting in giving us advice and encouragement while carrying out this research.
A The Code

```cpp
#ifndef _POINT_H
#define _POINT_H

template <unsigned int D>
class point {
public:
    __host__ __device__
    void set_coord(const unsigned int i, 
                   const float coord_i);
    __host__ __device__
    float const &
    get_coord(const unsigned int i) const;
    __host__ __device__
    float const &
    operator[](const unsigned int i) const;
    __host__ __device__
    unsigned int dimension() const
    {
        return D;
    }
    __host__ __device__
    float dot(const point &p) const;
    __host__ __device__
    point & difference(const point &p) const;
private:
    float m.coords[D];
};

struct measure{
    __host__ __device__
    measure();
    dataset<150,4> X;
    point<3> alpha;
    point<3> nu;
    point<4> medie1;
    point<4> medie2;
    point<4> medie3;
    covariance<4> covariance1;
    covariance<4> covariance2;
    covariance<4> covariance3;
    covariance<4> psi1;
    covariance<4> psi2;
    covariance<4> psi3;
    __host__ __device__
    double operator() (const point &beta) const;
};
```

Fig. 14: Point class.

Fig. 15: Measure struct.
template <unsigned int DD>
class covariance
{
    public:
        __host__ __device__
        unsigned int dimension() const
        { return DD; }
        
        //@ Returns element with no bound check (const version)
        //@ It allows a=m(1,1) on constant matrix m
        __host__ __device__
        float const & operator () (const unsigned int i, const unsigned int j) const
        { return data[i*DD+j]; };
        
        //@ Returns element with no bound check (non-const version)
        //@ It allows m(1,1)=1 on non-constant matrix m
        __host__ __device__
        float & operator () (const unsigned int i, const unsigned int j)
        { return data[i*DD+j]; };
        
        __host__ __device__
        covariance<DD> & operator=(const covariance<DD> & m);
        
        __host__
        void showMe(std::ostream & out=std::cout) const;
        
        __host__ __device__
        covariance<DD> sum(const covariance<DD> & m) const;
        
        __host__ __device__
        covariance<DD> product(const covariance<DD> & m) const;
        
        __host__ __device__
        covariance<DD> product(const float & alpha) const;
        
        __host__ __device__
        point<DD> product(const point<DD> & p) const;
        
        __host__ __device__
        float trace() const;
        
        __host__ __device__
        covariance<DD-1> algebraic_minor(const unsigned int i, const unsigned int j) const;
        
        __host__ __device__
        float determinant() const;
        
        __host__ __device__
        static covariance<DD> identity();
        
        __host__ __device__
        covariance<DD> inverse() const;

    private:
        float data[DD*DD];
};

Fig. 16: Covariance matrix class.
The Code

```
template <unsigned int D>
class hyperrectangle
{
public:
  __host__ __device__
  void set_bound(const unsigned int i, 
                 const float min, 
                 const float max);

  __host__ __device__
  thrust::pair<float, float> 
  get_bound(const unsigned int i) const;

  __host__ __device__
  thrust::pair<float, float> 
  operator[](const unsigned int i) const 
  { return get_bound(i); }

  __host__ __device__
  unsigned int dimension() const 
  { return D; }

private:
  float m_min[D], 
  m_max[D];
};
```
Fig. 17: Hyper-rectangle class.

```
template <unsigned int R, unsigned int C>
class dataset
{
public:
  __host__ __device__
  inline unsigned int nrow()const {return R;};

  __host__ __device__
  inline unsigned int ncol()const {return C;};

  __host__ __device__
  float const & operator ()(const unsigned int i, const unsigned int j) const 
  { return data[i*C+j];);

  __host__ __device__
  float & operator ()(const unsigned int i, const unsigned int j) 
  { return data[i*C+j];};

  __host__ __device__
  dataset<R,C> & operator=(const dataset<R,C> & my_dataset);

  __host__
  void showMe(std::ostream & out=std::cout) const;

  __host__ __device__
  point<C> diff_point_row(const unsigned int i, const point<C> & p) const;

private:
  float data[R*C];
};
```
Fig. 18: Dataset class.
The Code

```cpp
template <class M, unsigned int D>
struct allocator_on_global_type
{
  point<2*D> device_rect;
  thrust::device_vector<point<D>> beta_sampled;
  void allocate_device_rect(const hyperrectangle<D> & my_R);
  void allocate_beta_sampled(const unsigned int & num_th);
};

Fig. 19: Allocator on global memory struct.

```cpp
template <unsigned int D>
struct parallel_uniform_random
{
  point<2*D> my_rect;
  unsigned int my_num_threads;
  unsigned long long my_count_beta;
  unsigned long long my_count_u;
  unsigned int my_seed;

  __host__ __device__
  parallel_uniform_random(const point<2*D> & this_rect,
                           unsigned int num_th,
                           unsigned long long counter_u,
                           unsigned int this_seed);

  __host__ __device__
  point<D> operator[](unsigned int id);
};

Fig. 20: Parallel uniform random number generator struct.

```cpp
template <class M, unsigned int D>
struct is_smaller
{
  M my_measure;
  double my_u;

  __host__ __device__
  is_smaller(const M & this_measure, double this_u);

  __host__ __device__
  bool operator[](const point<D> & this_beta) const;
};

Fig. 21: Is smaller struct.
```
template <class M, unsigned int D>
class MCMC
{
    public:
        // CONSTRUCTORS
        MCMC<M,D>(){srand(time(NULL));};
        MCMC<M,D>(const hyperrectangle<D>& R);
        ~MCMC<M,D>(){};
    // SETTERS
    void set_hyperrectangle(const hyperrectangle<D>& R);
    void set_starting_point(const point<D>& p);
    std::vector<point<D>> > MakeChain(const M & measure,
                   const unsigned int G,
                   const unsigned int burn = 0,
                   const unsigned int thin = 0);
    void SaveChain (std::ostream &out, const std::vector<point<D>> &chain) const;
    private:
        M my_measure;
        hyperrectangle<D> R;
        double u;
        point<D> beta;
        void samplingU();
        void samplingBeta();
        unsigned int num_threads;
        unsigned long long counter_u;
        unsigned long long counter_beta;
        unsigned int m_seed;
        allocator_on_global_type<M,D> allocator_on_global;
        MCMC<M,D>(const MCMC<M,D> &);
        MCMC<M,D> & operator=(const MCMC<M,D> &);
};

Fig. 22: MCMC class, public and private members.
template <class M, unsigned int D>
void
MCMC<M,D>::samplingBeta()
{

typename thrust::device_vector<point<D> >::iterator end = allocator_on_global.beta_sampled.begin();

while( allocator_on_global.beta_sampled.begin()==end )
{

    thrust::transform(thrust::counting_iterator<int>({0}),
                       thrust::counting_iterator<int>({num_threads}),
                       allocator_on_global.beta_sampled.begin(),
                       parallel_uniform_random<D>(allocator_on_global.device_rect,
                       num_threads,
                       counter_beta,
                       counter_u,
                       m_seed));

    end = thrust::remove_if(allocator_on_global.beta_sampled.begin(),
                            allocator_on_global.beta_sampled.end(),
                            is_smaller<M,D>({my_measure,u}));

    ++counter_beta;
}

point<D> tmp(allocator_on_global.beta_sampled[0]);

    for(unsigned int i(0); i<D; ++i)
    
        beta.set_coord(i,tmp[i]);
}

Fig. 23: Sampling Beta algorithm.
template <class M, unsigned int D>
std::vector<point<D>> MCMC<M,D>::MakeChain(const M &measure,
const unsigned int G,
const unsigned int burn,
const unsigned int tin)
{
    std::cout << "Initializing Chain!" << std::endl;
    my_measure = measure;
    std::vector<point<D>> Chain;
    m_seed = rand();
    Chain.reserve(G);
    num_threads=48000;
    counter_u=0;
    counter_beta=0;
    unsigned int cols=0;
    allocator_on_local.allocate_device_rect(R);
    allocator_on_local.allocate_beta_sampled(num_threads);
    for(unsigned int i(0); i<burn; ++i)
    {
        samplingU();
        samplingBeta();
    }
    while(cols < G)
    {
        for(unsigned int i(0); i<tin; ++i)
        {
            samplingU();
            samplingBeta();
        }
        samplingU();
        samplingBeta();
        Chain.push_back(beta);
        cols++;
    }
    return Chain;
}

Fig. 24: Making Chain algorithm.