Politecnico di Milano

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING

Master of Science – Mathematical Engineering

Learning Block Structured Graphs in Gaussian Graphical Models

with an application to functional data analysis

Supervisor

Prof. Alessandra GUGLIELMI

Co-Supervisors

Prof. Lucia PACI

Prof. Alessia PINI

Candidate

Alessandro COLOMBI – 920383

Academic Year 2019 – 2020

A Bea

Sommario

I modelli grafici gaussiani sono un potente strumento statistico per lo studio della struttura di dipendenza fra le componenti di un vettore aleatorio. Tale struttura viene sintetizzata attraverso un grafo che però è incognito e quindi oggetto dell'inferenza statistica. Viene quindi incorporato in un modello gerarchico bayesiano, vincolando la struttura della matrice di precisione mediante una distribuzione a priori chiamata GWishart. Questo lavoro ha l'obbiettivo di estendere tali modelli. La novità introdotta consiste nell'imporre una struttura a blocchi alla matrice di adiacenza che descrive il grafo, al fine di semplificare interpretabilità della sua stima finale. A tal scopo, introduciamo una distribuzione a priori che vincola tale struttura. L'inferenza bayesiana è ottenuta sviluppando un algoritmo Markov Chain Monte Carlo trans-dimensionale, proponendo, ad ogni passo, di aggiungere o togliere un numero arbitrario di lati e non uno solo. In particolare, la procedura introdotta si basa su doppio salto reversibile evitando così la valutazione della costante di normalizzazione della GWishart. Il nuovo modello viene utilizzato per l'analisi di dati funzionali. In questo contesto, la classica tecnica di lisciamento mediante una base B-spline si arricchisce con lo studio della struttura di dipendenza dei coefficienti dell'espansione; infatti, il supporto compatto degli elementi della base trasferisce la dipendenza fra i coefficienti su porzioni ben precise del dominio delle funzioni lisciate. Inoltre, il modello permette lo scambio di informazioni tra le curve, grazie alla struttura gerarchica bayesiana. La maggior efficienza e interpretabilità del nostro modello rispetto a metodi alternativi è empiricamente verificata in diversi scenari simulati. Infine, il metodo proposto viene utilizzato per indagare la struttura di dipendenza tra bande dello spettro di assorbanza di puree di fragola. Tutti i modelli implementati sono disponibili in un pacchetto `R` chiamato Block Graph Structual Learning (`BGSL`).

Abstract

Gaussian graphical models are a powerful statistical tool to describe the concept of conditional independence between variables through a map between a graph and the family of multivariate normal models. The structure of the graph is unknown and has to be learned from data. Inference is carried out in a Bayesian framework: thus, the structure of the precision matrix is constrained by the graph through a GWishart prior distribution. In this work we aim to improve those models. Firstly, we introduce a prior distribution to impose a block structure in the adjacency matrix of the graph. Then we develop a Double Reversible Jumps Monte Carlo Markov chain that avoids any GWishart normalizing constant calculation when comparing graphical models. The novelty of this procedure is that it looks for block structured graphs, hence proposing moves that add or remove not just a single link but an entire group of them. The novel method is then applied to smooth functional data. The classical smoothing procedure is improved by placing a Gaussian graphical model on the basis expansion coefficients, providing an estimate of their conditional independence structure. Since the elements of a B-Spline basis have compact support, the independence structure is reflected on well defined portions of the domain. The Bayesian hierarchical formulation enables the borrowing of strength among different curves and the graphical model assumption allows to share information along different subintervals of the functional datum, which is possible since our model does not limit itself to block diagonal blocks, but it also admits long-term interactions. Through simulation studies, we discuss how our model improves efficiency and readability with respect to competitors. Finally, we learn the dependence structure among portions of the absorbance spectrum of strawberry purees. All implemented methods are available in our `R` package called Block Graph Structural Learning (`BGSL`).

Contents

Sommario v

Abstract vii

Contents xi

List of Figures xiv

List of Tables xv

Introduction 1

1 Gaussian Graphical Models 5

1.1 Introducing a space for Block graphs 8

1.2 Truncated Priors 10

1.3 Structural Learning 12

1.3.1 The `GWishart` normalizing constant 13

1.3.2 The marginal posterior probability of the graph 20

1.3.3 Birth and Death approach 23

2 Block Double Reversible Jumps for Gaussian Graphical Models 31

2.1 MCMC methods for Gaussian graphical models 32

2.2 Eliminating the posterior normalizing constant 36

2.2.1 Overview of Reversible Jumps MCMC 36

2.2.2 The Cholesky decomposition of `GWishart` random variates 38

2.2.3 Stochastic search in the complete graph space 40

2.2.4 Generalization to block graph space 44

2.3 Eliminating evaluation of prior normalizing constant 47

2.3.1 The Exchange algorithm 47

2.3.2 Double Reversible Jumps MCMC for GGM 50

2.3.3 Block Double Reversible Jumps MCMC for GGM 53

2.4 Comparison with similar frequentist approach 56

2.5 Application to functional data analysis 61

3 BGSL Package 67

3.1 External C++ libraries 69
3.1.1 The `sample` namespace 70
3.1.2 The `spline` namespace 74
3.1.3 The `HDF5conversion` namespace 76
3.2 Basic graphical objects implementation 81
3.2.1 Groups 82
3.2.2 Graphs 83
3.2.3 Precision matrix 88
3.3 `GWishart` computational aspects 90
3.3.1 A direct sampler for `GWishart` variates 90
3.3.2 Computing the normalizing constant 95
3.4 Implemented methods for Structural Learning 98
3.4.1 Graph priors 99
3.4.2 MCMC algorithms for GGM 101
3.5 `BGSL` samplers 108
3.5.1 `GGMsampler` 109
3.5.2 `FLMsampler` 112
3.5.3 `FGMsampler` 114
3.5.4 Posterior Inference 117
3.6 The `BGSL` `R` package 121
3.6.1 The `DESCRIPTION` file 121
3.6.2 The `NAMESPACE` file 123
3.6.3 Public interface: the exported functions 123
3.6.4 Compilation 125
3.6.5 Installation 126
3.6.6 How to use `BGSL` 128

4 Numerical Experiments 133

4.1 Validation of `GWishart` utilities 133
4.2 Testing GGMsampler 136
4.3 Testing FLMsampler 148
4.4 Testing FGMsampler 153
4.5 Real data application - fruit purees dataset 157

5 Discussion 165

Acronyms **169**

Ringraziamenti **177**

List of Figures

Figure 1.1 ρ map . . . 10
Figure 1.2 Truncated uniform prior . . . 11
Figure 1.3 Truncated Bernoulli prior . . . 11
Figure 1.4 Acceptance-rejection sampler . . . 13
Figure 1.5 Prior normalizing constant, low dimension . . . 18
Figure 1.6 Posterior normalizing constant, low dimension . . . 18
Figure 1.7 Prior and posterior normalizing constant, high dimension . . . 19
Figure 1.8 Birth and Death chain . . . 27

Figure 2.1 Add-Remove Metropolis Hastings chain . . . 34
Figure 2.2 Proposal distribution for block graphs . . . 35
Figure 2.3 Reversible Jumps chain . . . 38
Figure 2.4 Reversible Jumps for Gaussian graphical models chain . . . 43
Figure 2.5 Non scalability of ARMH . . . 55
Figure 2.6 k-component and bipartite graph . . . 60
Figure 2.7 Smoothing functional data . . . 61
Figure 2.8 B-spline basis representation . . . 64

Figure 3.1 HDF5 hyperslab selection . . . 78
Figure 3.2 Graph class hierarchy . . . 87
Figure 3.3 GGM polymorphic class hierarchy . . . 108

Figure 4.1 Graphs for low dimension tests . . . 135
Figure 4.2 Posterior normalizing constant, low dimension . . . 136
Figure 4.3 `GGMsampler`, experiment 1, estimated graphs . . . 138
Figure 4.4 `GGMsampler`, experiment 1, traceplots . . . 139
Figure 4.5 `GGMsampler`, experiment 1, estimated precision matrix . . . 140
Figure 4.6 `GGMsampler`, experiment 1, indices comparison . . . 140
Figure 4.7 `GGMsampler`, experiment 2 . . . 141
Figure 4.8 `GGMsampler`, experiment 3, estimated graphs . . . 143
Figure 4.9 `GGMsampler`, experiment 3, indices comparison . . . 145
Figure 4.10 `GGMsampler`, experiment 4, true graph and noisy precision matrix . . . 146

Figure 4.11 `GGMsampler`, experiment 4, scores . . . 147
Figure 4.12 `GGMsampler`, experiment 5 . . . 148
Figure 4.13 `FLMsampler`, functional datasets . . . 148
Figure 4.14 `FLMsampler`, smoothed curves . . . 149
Figure 4.15 `FLMsampler`, regression parameters . . . 150
Figure 4.16 `FLMsampler`, precision matrix . . . 151
Figure 4.17 `FLMsampler`, uncommon grids . . . 152
Figure 4.18 `FGMsampler`, functional datasets . . . 153
Figure 4.19 `FGMsampler`, estimated graphs, low dimension . . . 154
Figure 4.20 `FGMsampler`, traceplots, low dimension . . . 155
Figure 4.21 `FGMsampler`, estimated graph and traceplot, high dimension . . . 156
Figure 4.22 `FGMsampler`, precision matrix, high dimension . . . 156
Figure 4.23 Fruit purees dataset . . . 157
Figure 4.24 Fruit purees, smoothed curves . . . 159
Figure 4.25 Fruit purees, regression parameters . . . 159
Figure 4.26 Fruit purees, precision matrices . . . 160
Figure 4.27 Fruit purees, estimated graphs . . . 161
Figure 4.28 Fruit purees, different granularity levels . . . 163

List of Tables

Table 2.1 Notation for matrices involved in the DRJ chain 51

Table 3.1 Summary of implemented MCMC methods for GGM 102

Table 4.1 Benchmark comparison, `GWishart` exact sampler 135
Table 4.2 Benchmark comparison, `GWishart` normalizing constant 136
Table 4.3 `GGMsampler`, experiment 2, confusion matrices 142
Table 4.4 `GGMsampler`, experiment 3, confusion matrices 144
Table 4.5 Fruit purees, robustness test 163

Introduction

We are living in the era of data. Nowadays more and more decisions are data driven, we need data to understand the world around us, we want data to guide and tell us where to put our trust in and we wish we had enough data available to have predictions as reliable as possible. We do even trust data or indices we do not understand. We are anxious to collect them and to squeeze as much information as possible.

Within this paradigm of constantly improving our comprehension of the world through data, we here focus on performing statistical inference of dependence relationships among a large number of variables. This problem is rapidly gaining popularity as datasets become larger and data turn into big data. Our intuition suggests us that having more and more information at our disposal will immediately lead to a more faithful description of reality and a better prediction of future. This is true only if the underlying model is flexible enough to adapt itself and to constantly learn new features. For example, the standard approach in many situations is to model variables as independent. Once that this modeling choice is made, it does not matter how much information is available, the patterns of association between variables are not investigated and will not be discovered.

Our interest is instead in that part of the literature which tries to understand how variables interact with one another. One approach to this task is probabilistic graphical modeling (Lauritzen 1996). It relies on the concept of conditional independence between variables which is described through a map between a graph and a family of multivariate probability models. By doing so, their conditional independence structure can be directly read off from the graph. Its structure is usually unknown and has to be estimated from data. This process is called structural learning. When the family of probabilities is chosen to be Gaussian, we talk about Gaussian graphical models (GGM) .

This type of modelling appears in a huge variety of circumstances. The most prolific application field is probably the genetic one where those models come to the aid of biologists who want to infer the underlying genomic network between thousands of genes. Bhadra and Mallick (2013) and Mohammadi and Wit (2015) analysed gene expression data for humans. Their primary interest lain in finding the most significant associations between sets of single nucleotide polymorphisms (SNPs) and possibly correlated genetic transcripts. More genetic

examples are transcriptome expression variations from lymphoblastoid cell lines (Devijver and Gallopin 2018) and a botanical application on Arabidopsis thaliana described by Tan et al. (2015). Genetic analysis is particularly important in studying neoplasias. Jones et al. (2005); Dobra et al. (2004); Xia et al. (2018) studied expression data genes associated with the estrogen receptor pathway, whose role is one of the keys in the evolution and behaviour of breast cancer. Mohammadi and Wit (2015) also studied the mammary gland gene expression evolution in time. A further cancer genomics based example is given in Ni et al. (2019) for myelomas.
Other medical related problems that do not deal with genetics are, for example, a mapping of cancer mortality in the United States, whose purpose was an efficient resource allocation and planning, (Dobra et al. 2011), and a case study by Zhu et al. (2016); Qiao et al. (2019), who used functional data coming from electroencephalography in order to compare connections between areas of the brain in alcoholic and non alcoholic subjects.
Application in economics have been also explored in the recent years. In market basket analysis, Giudici and Castelo (2003) try to find relationships between a small number of purchases of individual customers. Scott and Carvalho (2008); Wang and Li (2012) search for patterns in mutual funds, motivated by the need for accurate estimates of variance and pairwise correlations of assets in dynamic portfolio selection problems. Researches were also done in studying the interconnectedness of credit risk. It is of particular interest in economic because it represents the risk that many financial institutions may fail together (Wang et al. 2015). There is actually no field of social sciences where graphical models could not perfectly fit in. Wild et al. (2010) reported a surge of network models being applied also to psychological datasets in recent years.

There are not well-defined boundaries on the range of their applicability, in principle one could use this type of modelling every time a Gaussian model is selected, removing the independence assumption among the variables of interest.
Despite the unquestionable potential to improve the modeling of data, the applicability of Gaussian graphical models is limited by the computational burden they bring along with them: for a $p-$dimensional variable there are in total $2^{\binom{p}{2}}$ possible conditional independence graphs. Even with a small number of variables, the model space is huge. The challenge is then being able to construct an efficient search algorithm that explores the graph space and that is able to learn what are the important edges and what are not.

Recovering the underlying network may not be enough, the interpretability of the results is the second challenge for a data scientist. The number of possible interactions grows with the number of variables at combinatorial speed, and so, the interpretation of the full network of dependencies becomes very challenging, if at all possible.

The focus of our research is on proposing a model extension to the classical GGM by enforcing a block structure in the adjacency matrix that describes the graph. The main idea

behind this assumption is that it is common to come across situations where the variables can be grouped and the structure of the most important dependencies can be synthesized by studying the relationships between and within those groups. In other words, the goal is no longer on looking for all possible relationships but on deriving the underlying pattern between groups.
For relatively small graphs, this assumption may look too restrictive, but, as soon as the number of variables grows, the importance of each possible dependence loses interest as it is more natural to try to understand the general structure.
The benefits of this procedure are twofold: it leads to results that are simpler to be interpreted and it provides sharper estimates because it enables a dimensionality reduction of the space where the graph is searched.

The Gaussian graphical models are usually incorporated into more complex problems. Functional data analysis is the application field that motivated our work.
In particular, we proposed a way to improve the classical smoothing procedure by placing a Gaussian graphical model on the coefficients of a B-Spline basis expansion, providing an estimate of their conditional independence structure. A B-Spline basis was selected because the compact support of its elements enables to reflect the independence structure on well-defined portions of the domain. The Bayesian hierarchical formulation enables the borrowing of strength among different curves and the graphical model assumption allows to share information along different subintervals of the functional datum, which is possible since the model does not limit itself to diagonal blocks, but it also admits long-term interactions.
A first approach to face this problem is described in Codazzi et al. (2021); our work provides an extension that considers the search for a block structured graph.

One of the goals of this work is to add a little tiny gear to this beautiful and complex machinery by developing an `R` package called `BGSL`, Block Graph Structural Learning.
We choose `R` (R Core Team 2013) because it guarantees a simple, user-friendly interface. Indeed, nowadays data analysis are so widespread that every program has to be suitable and accessible also for people having not so much confidence in advanced programming. `R` is perfect for that, its programming language is intuitive and easy to learn, it is uniform with respect to the choice of operating systems, which makes the results and code fully portable. It is a Free Software and its functionalities are constantly growing thanks to *packages*. Indeed every developer willing to share his code simply has to create his own package that can be immediately downloaded by all users. Everything is of course open source.

The drawback for so many comforts is efficiency. `R` language is interpreted, not compiled. Moreover, it is much slower than other interpreted languages such as MATLAB. Its weak spots include operations like nested loops, loops where the current computation depends on the previous one, accessing vector and matrix elements, recurrent functions and changing the size of vectors dynamically. Those are just some well-known examples, more can be found in

the R inferno, (Burns 2011), where an extensive and detailed analysis is given.
Those difficulties can be overcome thanks to R great flexibility. R environment allows to install a C++ compiler which makes it the perfect interface for compiled code and increases its efficiency. For computationally intensive tasks, C/C++ and even Fortran code can be linked and called at run time, while R language is more suitable for data manipulation and graphics.

BGSL fully exploits this dichotomy. All its computations are grounded on C++ code which is completely independent from R. The latter comes into play only as an interface for providing inputs and getting the outputs, so that the results are available for further statistical analysis. As every other package, BGSL is documented using R's LaTex-like documentation, that is generated using Roxygen2, (Wickham et al. 2020). Code is available in its own GitHub repository, https://github.com/alessandrocolombi/BGSL, where all instructions for installation are provided.

The dissertation is organized as follows:

Chapter 1: Gaussian Graphical Models

We introduce a Bayesian framework for Gaussian graphical models and present a new space for block graphs where we embed our Monte Carlo Markov chain. Moreover, we give a comprehensive analysis about the GWishart normalizing constant and to the Birth and Death approach for sampling from the posterior distribution of the graph and precision matrix.

Chapter 2: Block Double Reversible Jumps for Gaussian Graphical Models

This chapter is entirely dedicated to the presentation of our novel Monte Carlo Markov chain method for Gaussian graphical models. We then compare it with some similar existing models and finally we show its application to functional data analysis. It concludes the presentation of all theoretical aspects.

Chapter 3 : BGSL Package

We provide here all computational related aspects of our R package. First, we give an overview of the underlying C++ code, which is not expose to the user, and then we describe its public interface as well as how posterior inference of the sampled values is carried out.

Chapter 4 : Numerical Experiments

We conclude with an exhaustive presentation of BGSL capabilities. We test all available samplers on several simulation studies, providing experiments for multiple scenarios. The R package BDgraph is used for a challenging comparison of BGSL performance. We conclude with a case study about spectrometric data of fruit purees.

The final **Discussion** chapter draws some conclusions of this work summarizing the proposed novelties and discussing possible extensions.

Chapter 1

Gaussian Graphical Models

We start introducing some notations and definitions needed to frame the forecited undirected Gaussian graphical models. For a comprehensive introduction see Lauritzen (1996).

Let $G = (V, E)$ be an undirected graph of size p, where $V = \{1, \dots, p\}$ is the set of nodes, also called vertices, and $E \subset V \times V$ is the set of existing edges, or links. We call $\mathscr{G}$ the space of all possible graphs having p nodes. Self-loops are not allowed. Moreover, let

$$\mathscr{W} \; = \; \{ \; (i, j) \mid i, j \in V, \; i \leq j \; \} \tag{1.1}$$

be the set of possible links for graph G and $\bar{E} = \mathscr{W} \setminus E$ the set of its non-existing links, i.e all those edges that are compatible with V but that are not actually present in E.

We now introduce some graphical models notations that are used throughout this work. We only define those notions that are strictly necessary to our aim, see (Lauritzen 1996, Ch. 2) for detailed definitions and studies of these concepts.

An undirected graph $G = (V, E)$ is said to be *complete* if E contains all $\binom{p}{2}$ possible edges, otherwise it is said *incomplete*. We say that an incomplete graph G can be *decomposed* into disjoint subgraphs A, B and S if $G = A \cup B \cup S$ and S is a complete *separator*, which means that any path from a vertex in A to one in B goes through S. This decomposition is *proper* if neither A nor B is empty. Moreover, S is said to be minimal if it does not contain a proper subgraph that also separates A and B.

If the separator is always chosen to be minimal and G is iteratively decomposed, we end up defining its *prime components* P_i: a collection of subgraphs that cannot be further decomposed, i.e they are *maximal*.

If all the prime components of a graph are *cliques*, which means they are maximal and complete, the graph is said to be **decomposable**. In particular, when we refer to prime components that are cliques, we stress this property by calling them C_i rather than P_i.

Decomposable graphs have distributional properties that make them particularly tractable, but they form only a subset of all the possible ones, whose size is unknown. Bornn et al. (2011)

reports that in the literature there exists no result to calculate the number of decomposable graphs of a given size. For example, Wang and Li (2012) consider a 6-node graph, whose model space is small enough to be enumerated, hence it is possible to compute the proportion of decomposable graph, that is about 55% of the total. Nevertheless, it is not know how such proportion scales with respect to the dimension of the graph.

The literature about Gaussian graphical models first focused only on this kind of graphs, only recent theory developments allowed to enlarge the search for the solution to the whole space $\mathscr{G}$. In this case, we underline that the method is able to deal with **non decomposable** graphs.

Given $\boldsymbol{G} \in \mathscr{G}$, we define a zero mean **Gaussian graphical model**, GGM for short, with respect to graph $\boldsymbol{G}$ as

$$\mathscr{M}_G = \{ N_p(\mathbf{0}, \boldsymbol{K}) \mid \boldsymbol{K} \in \mathbb{P}_G \} \tag{1.2}$$

where $\mathbb{P}_G$ denotes the set of all $p \times p$ positive definite symmetric matrices whose structure is constrained by $\boldsymbol{G}$,

$$\mathbb{P}_G := \{ \boldsymbol{K} \text{ precision matrix, symmetric and positive definite} \mid k_{ij} = 0, \quad \forall (i,j) \notin E \}$$

Note that in (1.2) we indicated the multivariate normal distribution using the precision matrix instead of the covariance matrix. Unless specifically said otherwise, we would use this parametrization throughout this dissertation.
Such notation is useful because it allows to read from the graph the Markov property that characterizes the random variables in model (1.2). To be more precise, given an undirected graph $\boldsymbol{G} = (V, E)$, a random vector $\boldsymbol{y} = \{y_u\}_{u \in V}$ indexed by V, form a **Markov Random Field** with respect to $\boldsymbol{G}$ if every possible pair of its elements is conditionally independent. Namely, if

$$y_i \perp\!\!\!\perp y_j \mid \boldsymbol{y}_{V \setminus \{i,j\}}, \qquad i \neq j \tag{1.3}$$

where $\boldsymbol{y}_{V \setminus \{i,j\}}$ is the random vector containing all elements in $\boldsymbol{y}$ except the i-th and the j-th. Given the normality hypothesis, for model (1.2) the following are all equivalent (Wang et al. 2015)

$$y_i \perp\!\!\!\perp y_j \mid \boldsymbol{y}_{V \setminus \{i,j\}} \iff G_{ij} = 0 \iff (i,j) \notin E \iff k_{ij} = 0 \tag{1.4}$$

where G_{ij} is an abuse of notation to identify the graph with its adjacency matrix that is a $p \times p$ symmetric matrix where $G_{ij} = 1$ or 0 whether the link (i,j) belongs to E or not. Relationship (1.4) clarifies the bound between the conditional independence structure for the random variables, the graph and the precision matrix.

Let $\boldsymbol{Y} = \{\boldsymbol{y_1}, \dots, \boldsymbol{y}_n\}$, $\boldsymbol{y_i} \in \mathbb{R}^p$, $\forall i = 1 : p$, be an independent and identically distributed (iid) sample of size n from model (1.2).

The corresponding likelihood is

$$P(\boldsymbol{Y} \mid \boldsymbol{K}, G) \;\propto\; |\boldsymbol{K}|^{n/2} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K} \sum_{i=1}^{n} \boldsymbol{y}_i \boldsymbol{y}_i^T\right)\right\} \tag{1.5}$$

To work in a Bayesian framework, we need to specify *prior distributions* for both parameters, $\boldsymbol{G}$ and $\boldsymbol{K}$. For the prior of the precision matrix, we focus on that part of the literature that chooses a GWishart (Roverato 2002) which is attractive since it represents the conjugate prior for normally distributed data. Following Mohammadi and Wit (2015), we prefer to use a "Shape-Inverse Scale" parametrization, even though it was not the one use in Roverato (2002). Given $\boldsymbol{G} \in \mathscr{G}$, a random matrix $\boldsymbol{K} \in \mathbb{P}_G$ follows a GWishart distribution if its density is

$$P\left(\boldsymbol{K} \mid G, b, D\right) = I_G\left(b, D\right)^{-1} |\boldsymbol{K}|^{\frac{b-2}{2}} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K} D\right)\right\} \tag{1.6}$$

In this case we write $\boldsymbol{K} \sim \text{GWishart}(b, \boldsymbol{D})$, where $b > 2$ is the Shape parameter and the Inverse Scale matrix $\boldsymbol{D}$ is symmetric and positive definite. Those quantities are usually hyperparameters, i.e they are fixed, not random.
$I_G(b, \boldsymbol{D})$ is the normalizing constant,

$$I_G(b, D) = \int_{\mathbb{P}_G} |\boldsymbol{K}|^{\frac{b-2}{2}} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K} D\right)\right\} \, d\boldsymbol{K} \tag{1.7}$$

We have to leave it indicated in integral form because, in general, that integral can not be computed to get an explicit analytical formula.
As mentioned before, prior (1.6) is conjugated to the likelihood (1.5), hence, given $\boldsymbol{G}$ and observed data $\boldsymbol{Y}$, a posteriori the random precision matrix is distributed as

$$\boldsymbol{K} \mid G, \boldsymbol{Y} \;\sim\; \text{GWishart}(b + n, D + U) \tag{1.8}$$

where $U = \sum_{i=1}^{n} \boldsymbol{y}_i \boldsymbol{y}_i^T$.
We can finally introduce the Bayesian hierachical formulation for model (1.2),

$$\begin{aligned} \boldsymbol{y}_1, \dots, \boldsymbol{y}_n \mid \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\boldsymbol{0}, \; \boldsymbol{K}\right) \\ \boldsymbol{K} \mid G &\sim \text{GWishart}\left(b, D\right) \\ G &\sim \pi\left(G\right) \end{aligned} \tag{1.9}$$

In the rest of the work, we always refer to a GGM in its Bayesian hierarchical formulation (1.9).

1.1 Introducing a space for Block graphs

Given model (1.9), being able to recover the true graph G from data is very challenging. In Bayesian statistics, it is considered a random variable living in the space $\mathscr{G}$, which is a discrete space. To draw samples from the marginal posterior of G, in principle, we should first recover the distribution, that implies computing $P(G \mid Y)$ $\forall G \in \mathscr{G}$ and then choosing the graph proportionally to the computed probabilities. It is clear that this approach is not feasible due to the magnitude of space $\mathscr{G}$. Its cardinality grows at combinatorial speed with respect to the number of nodes, $|\mathscr{G}|= 2^{\binom{p}{2}}$. Just to have an idea of how big it can be, just note that $p = 7$ implies more than 2 million possible graphs. Hence, in practice it is not possible to enumerate all graphs, the only possible way to navigate this space is by means of a Monte Carlo Markov chain method, MCMC for short.

Several methods have been proposed to explore this space, we review the most important ones in Chapter 2 where we also propose a new technique. Before entering in details, we need to lay down some theoretical definitions.
The starting point for our work is that, even though there exist very sophisticated MCMC methods to explore $\mathscr{G}$, this space is so big that some edges are always wrongly classified. At first, the literature mainly focused on exploring only a subset, the one composed only of decomposable graphs, previously defined. This choice, however, was forced by computational reasons. As we will see, those graphs allow to derive an explicit analytical formula for the GWishart normalizing constant $I_G(b, D)$, see definition (1.7). The development of the theory and increasing computation power paved the way for removing this restrictive hypothesis, but, as just said, being able to fully explore that space implies the possibility for a search method to get lost into it or simply to explore regions that could be non informative.
What we want to do is to combine those aspects, we want to restrict our MCMC to move only in a subset of $\mathscr{G}$ but we want this space reduction to be taken according to the prior available knowledge of the problem and not because of computational limitations.

In particular, what we want to look for is a graph having a block structure. Suppose to deal with a problem such that it is known, a priori, that the variables of interest can be grouped in smaller subsets. We focus only on searching interactions between those groups. Within interactions are also taken into account, in the sense that we only want to estimate if all variable are interacting one another or if there are no dependencies at all. The result may look more coarse, but it is much easier to be interpreted. Indeed, as the number of variables into play grows, it is useless to look for all possible interactions and it is instead more interesting to search for more meaningful relationships which may suggest a pattern and allow to look at the problem from a wider prospective.

Block graph representation

First of all, we need to introduce the appropriate definitions for this new space of block graphs. Consider $G = (V, E) \in \mathcal{G}$ and define $G_A = (V_A, E_A)$ such that $V_A = \{A_1, ..., A_M\}$ is a partition of V and each A_i, called group, has cardinality n_i, $\forall i = 1, ..., M$. We simply call $|V_A| = M$, that is the number of selected groups.
Similarly as before $\mathcal{W}_A = \{ \ (l, m) \mid l, m \in V_A, \ l \leq m \ \}$ is the set of all possible links connecting the M groups.
To ensure a meaningful structure, we introduce the following addition hypothesis,

$$\text{Given } i \in A_k, \ i > j \ \forall j \in A_h, \ \forall h < k. \tag{1.10}$$

which has to be true $\forall i \in A_k$ and $\forall k = 1, ..., M$. It means that all indexes that belong to A_k have to be strictly greater than all indexes in all previous groups. If $n_i > 1$ we also allow the possibility of having self loops, which implies that G_A is a *multigraph* rather than a simple graph. We also take into consideration groups whose cardinality may be just 1, in this case we call them *singletons*. We do not allow self loops for them.
Given V_A, we define $\mathcal{G}_A$ the set of all possible multigraphs having V_A as set of nodes. In the following, we want to clarify the relationship between that space and $\mathcal{G}$.

Consider $G_A \in \mathcal{G}_A$ and $G \in \mathcal{G}$. By definition, the set of nodes of the first (multi)graph is obtained by grouping together the nodes of the second graph. What about the set of edges? Are the two sets somehow connected? We want to create a map which defines a relationship between them.
Define $\rho : \mathcal{G}_A \longrightarrow \mathcal{G}$, such that $G_A = (V_A, E_A) \mapsto G = (V, E)$ by the following transformations

$$\begin{aligned} &V = \{1, \ldots, p\} \\ &\text{if } (l, m) \in E_A \Rightarrow \ (i, j) \in E \ \forall i \in A_l, \ \forall j \in A_m \\ &\text{if } (l, m) \notin E_A \Rightarrow \ (i, j) \notin E \ \forall i \in A_l, \ \forall j \in A_m \end{aligned} \tag{1.11}$$

Note that the definitions make sense even if $m = l$ and also if n_l or n_m are equal to 1. Indeed, every G in $\mathcal{G}$ does not admit self loops and every G_A in $\mathcal{G}_A$ does not admit self loops if $n_l = 1$.
A visual representation of this mapping is given in Figure 1.1.
Once ρ is set we are able to associate each G_A in $\mathcal{G}_A$ to one and only one G in $\mathcal{G}$ since ρ is clearly injective. We refer to G_A as the block form of G.

Nevertheless, ρ is not surjective which implies that there are graphs that do not have a representative in $\mathcal{G}_A$. As the name suggests, only those graphs with a particular block structure, can be represented in block form. Non surjectivity of ρ is obvious just looking at the cardinality of the spaces, $|\mathcal{G}_A| = 2^{\binom{M}{2} + M - \sum_{i=1}^{M} \mathbf{1}_{n_i = 1}}$ is in general less than $|\mathcal{G}| = 2^{\binom{p}{2}}$. Just

Figure 1.1. The map from multigraph G_A (left) to its complete form G (right)

to visualize the difference, if $p = 6$ there are already 32768 possible graphs in $\mathscr{G}$ but choosing $M = 3$ and $n_1 = n_2 = n_3 = 2$ there are only $2^6 = 64$ graphs in $\mathscr{G}_A$.
A non surjective map is the key ingredient for navigating only a subset of $\mathscr{G}$. Let us consider the range of ρ, denoted by $\mathscr{B}$. It is the subset of $\mathscr{G}$ containing all the graphs having p nodes and a block structure induced by ρ. Moreover, $\rho : \mathscr{G}_A \to \mathscr{B}$ is a bijection, which means that every graph $G \in \mathscr{B}$ is associated to its representative $G_A \in \mathscr{G}_A$ via ρ^{-1}. We say that $G \in \mathscr{B}$ is the complete representation of the (multi)graph $G_A \in \mathscr{G}_A$.

Given model (1.9), our idea is to restrict our search for G only to $\mathscr{B}$ and it can be done by exploiting functions ρ and ρ^{-1} to switch back and forth from the complete representations to their block forms.

1.2 Truncated Priors

Since we would like to move among graphs in $\mathscr{B}$, we may think of a prior π in model (1.9) that places zero mass probability on all those graphs that belong to $\mathscr{G} \backslash \mathscr{B}$. We call those priors *truncated priors*. We here focus on two possibilities, *Uniform* and *Bernoulli*.
We here stress the difference between G that in the model represents a random variable having values in $\mathscr{G}$ and $g \in \mathscr{G}$ that is a realization of G. Only $g = (V, E)$ is actually a graph.

- **Truncated-Uniform Prior**

$$\pi_{tu}(G = g) = \begin{cases} \dfrac{1}{|\mathscr{G}_A|} & \text{if } g \in \mathscr{B} \\ 0 & \text{if } g \in \mathscr{G} \backslash \mathscr{B} \end{cases} \tag{1.12}$$

This prior constrains the random variable G to take values only in $\mathscr{B}$ and it is non informative in that particular space. Figure 1.2 is a simple example for the case $p = 3$, $M = 2$, $n_1 = 2$ and $n_2 = 1$. There are eight possible complete graphs but only four of them have a representative in $\mathscr{G}_A$. We set a non-informative, uniform prior in that space which is then mapped on the corresponding complete graphs.

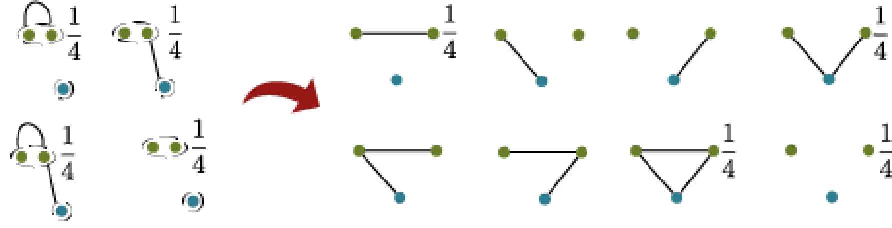

Figure 1.2. Representation of the Truncated Uniform prior. Here p is equal to 3, $M = 2$ and $n_1 = 1$, $n_2 = 2$. There are then only 4 possible graph in block form. We place a uniform probability distribution on the graphs in $\mathcal{G}_A$ (left) that is mapped in $\mathcal{G}$ (right). $|\mathcal{G} \backslash \mathcal{B}|$ is equal to 4 and indeed there are four graphs having zero probability mass.

- **Truncated-Bernoulli Prior** Alternatively, it is possible to induce a prior on the graph by assigning independent Bernoulli priors on each possible link, $\forall e \in \mathcal{W}$, $P(e \in E) = \theta_e \in (0, 1)$. Once the probability of every edge is set, the probability of each graph $g = (V, E)$ is simply proportional to the realization on each link in $\mathcal{W}$, that is

$$\pi_b(G = g) \propto \prod_{e \in E} \theta_e \prod_{e \in \bar{E}} (1 - \theta_e) \propto \theta^{|E|} (1 - \theta)^{\binom{p}{2} - |E|} \quad \forall g \in \mathcal{G} \tag{1.13}$$

In (1.13), for the sake of simplicity, we have considered the case $\theta_e = \theta$, $\forall e \in \mathcal{W}$. π_b defined in (1.13) is a valid prior but it is not truncated, indeed it places positive probability on every possible graph. We would like to set up a very similar prior but in $\mathcal{B}$. This procedure requires two steps.
First of all, we define the Bernoulli prior on $\mathcal{G}_A$. Then we map all the variables in $\mathcal{B}$ and set to 0 the probability of each graph outside that space.
Taking into account that there is a one-to-one relationship between $\mathcal{B}$ and $\mathcal{G}_A$, the following is straightforward:

$$\pi_{tb}(G = g) = \begin{cases} \pi_b(\rho^{-1}(G) = \rho^{-1}(g)) & \text{if } g \in \mathcal{B} \\ 0 & \text{if } g \in \mathcal{G} \backslash \mathcal{B} \end{cases} \tag{1.14}$$

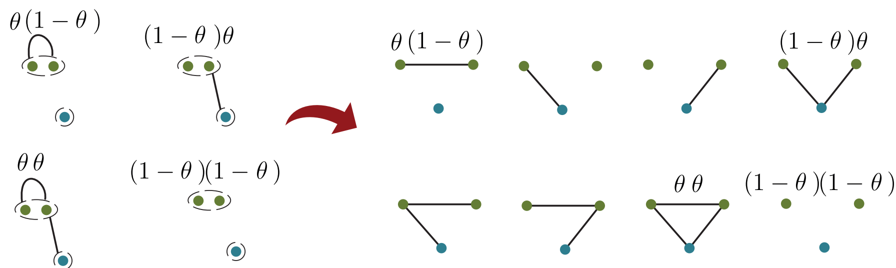

Figure 1.3. Representation of the Truncated Bernoulli prior. We first define the probability of each possible block form, in this case every possible edge in this space has probability θ, and then it is mapped into the complete graph space as is Figure 1.2.

A simple representation is given in Figure 1.3. As the previous example, there are 8 possible complete graphs and only 4 block graphs. The probability of inclusion for their link is θ and the resulting probability is then mapped on the corresponding complete graphs.

In general, it is always possible to set a truncated prior on $\mathscr{G}$ simply by assigning a prior on $\mathscr{G}_A$, whose dimension is much smaller. Then it is mapped into $\mathscr{B}$ and completed by assigning 0 everywhere else.

1.3 Structural Learning

The underlying graph is usually unknown and it has to be learned from data. This type of problem, called **Structural learning**, is not new to the literature. The goal of this section is to provide an overview of the most common methods currently available and to highlight those aspects that may or may not be well suited for a generalization to block structured graphs. We indeed underline that all methods we are aware of look for a solution in the space of all possible graphs $\mathscr{G}$, none of them is able to force the solution to belong to $\mathscr{B}$, that is the space of graphs having a block structure.
We now focus on the Bayesian statistics way of approaching this problem, a glance to existing frequentist methodologies is given in Section 2.4.

As already mentioned, the source of all troubles in structural learning problems is the cardinality of $\mathscr{G}$. Given a p-dimensional Gaussian graphical model, there are $2^{\binom{p}{2}}$ possible graphs that may represent its conditional dependence structure. Just to give an idea, if we choose p to be equal to 7, we end up having more than 2 million possible graphs. Since the dimension of this space grows at combinatorial speed, it is clear that there is no concrete possibility to enumerate and compare all possible models. The only way to traverse the set $\mathscr{G}$ is by means of a Monte Carlo Markov chain method.

A typical way to set up this chains is by means of a rejection algorithm. Since both the graph $\boldsymbol{G}$ and the precision matrix $\boldsymbol{K}$ are unknown parameters, let us denote with $(\boldsymbol{K}^{[s]}, \boldsymbol{G}^{[s]})$ the current state of the chain. The subsequent state is defined by the following steps: first of all propose a new couple of parameters, let us say $(\boldsymbol{K}', \boldsymbol{G}')$, drawn from a certain proposal distribution. The model defined according to the current state and the proposed one and then compared to obtain the acceptance rejection probability γ. The new state $(\boldsymbol{K}^{[s+1]}, \boldsymbol{G}^{[s+1]})$ is set equal to $(\boldsymbol{K}', \boldsymbol{G}')$ with probability γ or it remains equal to the old one with probability $1 - \gamma$.
Within this family of Markov chains several possibilities are available, depending on how the proposed state is generated and the way the two models are compared. We can indeed distinguish two subfamilies of models, the ones that work with the marginal posterior distribution of the graph and the ones that instead build a chain on the joint space of precision matrix

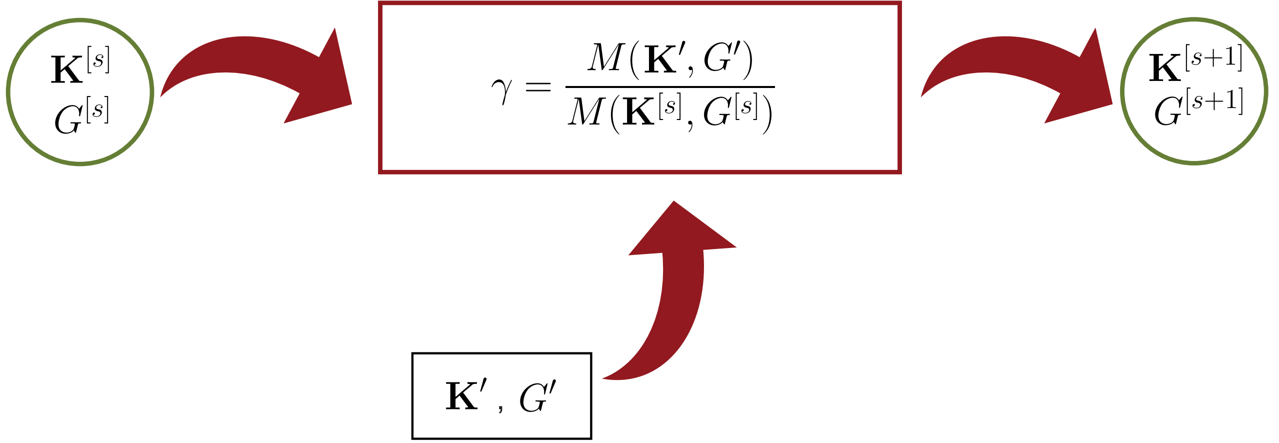

Figure 1.4. Visual representation of a general acceptance-rejection Monte Carlo Markov chain on the joint space of possible graphs and precision matrix. $(\boldsymbol{K}^{[s]}, \boldsymbol{G}^{[s]})$ is the current state of the chain, $(\boldsymbol{K}', \boldsymbol{G}')$ is the proposed one and $(\boldsymbol{K}^{[s+1]}, \boldsymbol{G}^{[s+1]})$ the subsequent one, which would be equal to the old state with probability $1-\gamma$ or to the proposed one with probability γ.

and graph. Some examples are analyzed in Section 1.3.2.
Rejection algorithms are not the only available choice, recently Mohammadi and Wit (2015) successfully developed a *Birth and Death* approach that has the marvelous property of accepting all proposed moves, which makes it a very efficient method. Unfortunately its generalization to the block graph case is very challenging, since it is specifically designed to modify only one link at a time. More details are given in Section 1.3.3.

Before entering the details of all those MCMC methods, we want to give an exhaustive description of the normalizing constant for a GWishart distribution, as its role is crucial in the development of this work.

1.3.1 The `GWishart` normalizing constant

As already mentioned, we focus on that part of the literature that chooses a GWishart prior for the precision matrix. As stated at the beginning of this chapter, we follow a "Shape-Inverse Scale" parametrization for this distribution. For the sake of clarity, we report its density here below,

$$P\left(\boldsymbol{K} \mid G, b, D\right) = I_G\left(b, D\right)^{-1} |\boldsymbol{K}|^{\frac{b-2}{2}} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K} D\right)\right\}$$

where $I_G(b, D)$ is the normalizing constant,

$$I_G(b, D) = \int_{\mathbb{P}_G} |\boldsymbol{K}|^{\frac{b-2}{2}} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K} D\right)\right\} \, d\boldsymbol{K}$$

which plays a central role in the computation of the Bayes factors in model comparison as well as in model selection problems as it is shown later on. For those reasons, it is crucial to

calculate it precisely.
When the graph is complete, the GWishart distribution simply collapse to be a Wishart distribution. In this case the computation of $I_G(b, D)$ is harmless, as it reduces to

$$I_G(b, D) = \frac{2^{(b+p-1)p/2}\, \Gamma_p\{(b+p-1)/2\}}{|D|^{(b+p-1)/2}} \tag{1.15}$$

where $\Gamma_p(m)$ is the multivariate gamma distribution, defined, for $m > (p-1)/2$, as

$$\Gamma_p(m) = \pi^{p(p-1)/4} \prod_{i=0}^{p-1} \Gamma\left(m - \frac{i}{2}\right)$$

Another case when this constant can be easily computed is when G is decomposable. Let $(C_j, j = 1, \ldots, k)$ and $(S_j, j = 2, \ldots, j)$ denote a perfect ordering of the set of cliques of G and its corresponding set of separators having cardinality c_j and s_j. Roverato (2002) and Atay-Kayis and Massam (2005) proved that the normalizing constant can be factorized as

$$I_G(b, D) = \frac{\prod_{j=1}^{k} I_{C_j}(b, D_{C_j})}{\prod_{j=2}^{k} I_{S_j}(b, D_{S_j})} \tag{1.16}$$

where I_{C_j} and I_{S_j} are the normalizing constants referred to subgraphs C_j, for $j = 1, \ldots, k$, and S_j, for $j = 2, \ldots, k$, that are all complete and can therefore be computed using (1.15).

The non decomposable case is instead much more difficult to handle. An exact analytic expression for computing $I_G(b, D)$ in this general case has actually been derived by Uhler et al. (2018), however, as noted in Letac et al. (2017), it is mathematically complex and at present time impossible to implement practically. This constant is then untractable, it can only be computed numerically. We are aware of four different methods to do that: Roverato (2002) and Dellaportas et al. (2003) developed an importance sampler, Lenkoski and Dobra (2011) addressed the problem by proposing a Laplace transform approximation. However, the reference method, the one that is mainly used in the literature and the one we will refer to throughout this dissertation is the Monte Carlo approximation developed by Atay-Kayis and Massam (2005). We briefly describe it here below.

Given $G = (V, E)$, consider $\boldsymbol{K} \mid G \sim \text{GWishart}(b, D)$, $\boldsymbol{K} \in \mathbb{P}_G$. The method we are presenting is based on two subsequent changes of variables.
The first step is to compute the Cholesky decomposition of $\boldsymbol{K}$, which is always possible as it is symmetric and positive definite. Let $\boldsymbol{K} = \Phi^T \Phi$ where Φ is an upper triangular matrix. Then, as an intermediate step, one has to factorize the Inverse Scale matrix D as $D = (T^T T)^{-1}$, where T is an upper triangular matrix representing the Cholesky decomposition of D^{-1}. The second change of variable consists of defining $\psi = \Phi T^{-1}$.

To understand the rationale behind this double transformation, we need some further definitions. Let $\nu(G) = \{(i, j) : i = j \text{ and } i \in V \text{ or } (i, j) \in E\}$ the set of the diagonal elements and the links belonging to G, we refer to $\psi^{\nu(G)} = \{\psi_{ij} : (i, j) \in \nu(G)\}$ as the set of *free elements of* ψ, whose name is due to the fact that all other elements of ψ, i.e all those ψ_{ij} such that $i, j \in V$ but $(i, j) \notin \nu(G)$, can be derived as a function of the entries in $\psi^{\nu(G)}$. Their relationship has been proved in (Atay-Kayis and Massam 2005, Lemma 2).

Theorem 1.3.1.1. *Let $G = (V, E)$ and $K \in \mathbb{P}_G$. Moreover, let $\boldsymbol{K} = \Phi^T\Phi$ be its Cholesky decomposition, where Φ is an upper triangular matrix. Let T be a given upper triangular matrix such that $\psi = \Phi T^{-1}$. Define a further upper triangular matrix $\boldsymbol{H}$ such that $h_{ij} = t_{ij}/t_{jj}$.*
For every $(i, j) \in \nu(G)$ we have that

$$\psi_{ij} = \sum_{k=i}^{j-1} -\psi_{ik} h_{kj} + \frac{\phi_{ij}}{t_{jj}} \tag{1.17}$$

In particular, for $i = j$,

$$\psi_{jj} = \frac{\phi_{jj}}{t_{jj}} \tag{1.18}$$

Instead, for $(i, j) \notin \nu(G)$, $i < j$

$$\psi_{ij} = -\sum_{k=1}^{j-1} \psi_{ik}\ h_{kj} - \sum_{r=1}^{i-1} \left(\frac{\psi_{ri} + \sum_{l=r}^{i-1} \psi_{rl}\ h_{li}}{\psi_{ii}} \right) \left(\psi_{rj} + \sum_{l=r}^{j-1} \psi_{rl}\ h_{lj} \right) \tag{1.19}$$

In particular, if $i = 1$, for all $j > 1$ such that $(1, j) \notin \nu(G)$

$$\psi_{ij} = -\sum_{k=1}^{j-1} \psi_{ik}\ h_{kj} \tag{1.20}$$

Equations (1.17) and (1.18) clarify the reason why $\psi^{\nu(G)}$ are called free elements, as their values explicitly depend on Φ, which is uniquely determined by $\boldsymbol{K}$ that is fixed within this context. This is no longer true in (1.19) and (1.20), in this case ψ_{ij} is determined by all those elements ψ_{rs} such that $(r, s) \overset{L}{<} (i, j)$, where $\overset{L}{<}$ represents lexicographic compare. This implies that, given $\psi^{\nu(G)}$, all the remaining entries can be found recursively following the lexicographic order.

This partition of the elements of ψ is exploited in the Monte Carlo approximation of $I_G(b, D)$. The latter can indeed be expressed as the product of a constant term and the expectation of a function of the nonfree elements, that are in turn functions of the free one, as explained in Theorem 1.3.1.1. A Monte Carlo method can then be applied as $\psi^{\nu(G)}$ can be easily generated since the diagonal elements follows a χ^2 distribution and the others a standard normal distribution (Roverato 2002).

Such decomposition is reported in Theorem 1.3.1.2 (Atay-Kayis and Massam 2005, Theorem 1) but before stating it, we need to introduce

$$\nu_i^G = |\{j : j > i \text{ and } (i,j) \in E\}| = \ |N_i \cap \{i+1, ..., p\}| \tag{1.21}$$

For $i = 1, ..., p-1$. It is equal to the number of 1s in row i, from position $i+1$ up to the end. For convention, $\nu_p^G = 0$.

Theorem 1.3.1.2. *Let G be an arbitrary undirected, non decomposable, graph and let $I_G(b,D)$ be the normalizing constant of the* GWishart(b,D) *as defined in* (1.6)*. Then,*

$$I_G(b,D) = \prod_{i=1}^{p} \left((2\pi)^{\nu_i^G/2} \ 2^{(b+\nu_i^G)/2} \ \Gamma\left(\frac{b+\nu_i^G}{2}\right) \ (t_{ii})^{b+\nu_i^G} \right) \mathbb{E}[f_T(\psi^{\nu(G)})] \tag{1.22}$$

where

$$f_T(\psi^{\nu(G)})) = \exp\left\{ -\frac{1}{2} \sum_{(i,j)\notin\nu(G)} \psi_{ij}^2 \right\} \tag{1.23}$$

and the elements ψ_{ij} within (1.23) *are well defined functions of $\psi^{\nu(G)}$ and stated in Theorem 1.3.1.1.*
Moreover, the expected value in (1.22) *is taken with respect to the distribution with density equal to the product of independent chi-squared distributions with $b + |\nu_i^G|$ degree of freedom and standard normal distributions. More precisely,*

$$\psi_{ii} \overset{\text{ind}}{\sim} \sqrt{\chi^2(b + |\nu_i^G|)}, \qquad i = 1, \ldots, p \tag{1.24}$$

$$\psi_{ij} \overset{\text{iid}}{\sim} N(0,1), \qquad (i,j) \in E \tag{1.25}$$

The resulting algorithm is reported in Algorithm 1, some computational aspects are analyzed in Section 3.3.2.

Thanks to Theorem 1.3.1.2 we are capable of approximating the value of $I_G(b,D)$ also for non decomposable graphs. Unfortunately, there is not a theoretical valid estimate of its Monte Carlo variance which would be needed to decide the number N of iterations to be performed. In practice, such variance may result to be very high which makes this method unreliable in high dimensional problems, as noted in Jones et al. (2005); Mohammadi and Wit (2015); Wang and Li (2012).Test performed by means of our implementation of Algorithm 1 highlighted the same problems noticed in the previously mentioned works.
Throughout this dissertation, we refer to $I_G(b,D)$ by calling it **prior normalizing constant**, since b and D are the prior hyperparameters for $\boldsymbol{K}$ in model (1.9). The GWishart prior is conjugate to the likelihood (1.5) and its posterior distribution is reported in (1.8). The normalizing constant for that particular distribution is indicated by $I_G(b+n, D+U)$ and we

Algorithm 1: Monte Carlo approximation for $I_G(b, D)$

Step 1. Compute D^{-1} its Cholesky factorization T such that $D^{-1} = T^T T$.

Step 2. Define $h_{ij} = t_{ij}/t_{jj}$ for $1 \le i \le j \le p$.

Step 3. Let N be the number of Monte Carlo iterations, for iter = $1, ..., N$ repeat the following steps:

Step 3.1 Draw the free elements as stated in (1.24) and (1.25).

Step 3.2 For $1 \le i \le j \le p$ and $(i, j) \notin E$, derive the remaining elements of ψ as explained in (1.20) and (1.19)

Step 3.3 Compute

$$\mathrm{J(iter)} = \exp\left\{-\frac{1}{2}\sum_{(ij)\notin E}\left(\psi_{ij}\right)^2\right\} \tag{1.26}$$

Step 4. Compute the Monte Carlo mean

$$\hat{\mathrm{J}}_{MC} = \frac{1}{N}\sum_{\mathrm{iter}=1}^{N}\mathrm{J(iter)} \tag{1.27}$$

Step 5. Compute the constant term C_G

$$C_G = \prod_{i=1}^{p}\left((2\pi)^{\nu_i^G/2}\; 2^{(b+\nu_i^G)/2}\;\Gamma\left(\frac{b+\nu_i^G}{2}\right)\;(t_{ii})^{b+\nu_i^G}\right) \tag{1.28}$$

Step 6. Finally, $I_G(b, D)$ is approximated by

$$\hat{I}_G^{MC} = C_G\;\hat{\mathrm{J}}_{MC}$$

---

refer to it as **posterior normalizing constant**. The reason why we use different names to distinguish those quantities is that the second one is usually characterized by higher variance, or, from another point of view, the classical choice $D = I_p$ reduces the variability of the prior normalizing constant.
The following examples are aimed to demonstrate two major difficulties,

(a) The Monte Carlo variance is higher when computing the posterior normalizing constant with respect to the prior normalizing one, given the choice $D = I_p$.

(b) Estimates get worse as the number of nodes p increases.

In practice this implies that we are able to get precise estimates only in low dimensional cases. In the experiment displayed in Figure 1.5, we fixed $p = 6$ and we computed the empirical

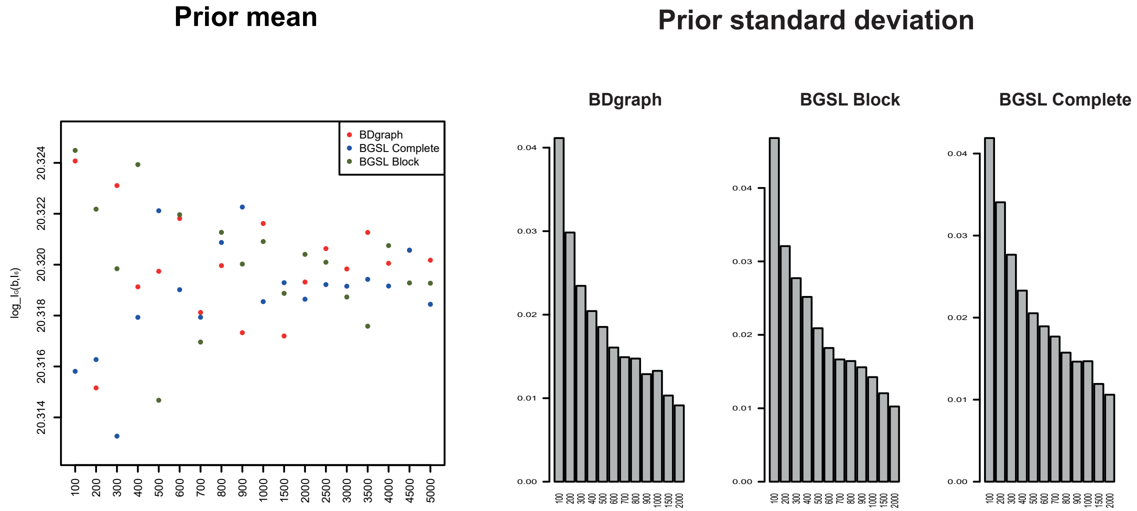

Figure 1.5. Monte Carlo empirical mean for prior normalizing constant (left) and the corresponding standard deviation (right). The underlying graph has $p = 6$ nodes. Plots are given in logarithmic scale, obtain with 500 repetitions for each possible choice of Monte carlo iterations.

mean and standard deviation of a prior normalizing constant $I_G(3, \boldsymbol{I}_6)$, out of 500 repetitions for different choices of the number of Monte Carlo iterations.

In our R package BGSL we provide a function that implements Algorithm 1. We compare ourselves to the analogous function of the R package BDgraph. The observed differences in the outputs seem to be related to the intrinsic variability of the method, no systematic errors were observed.

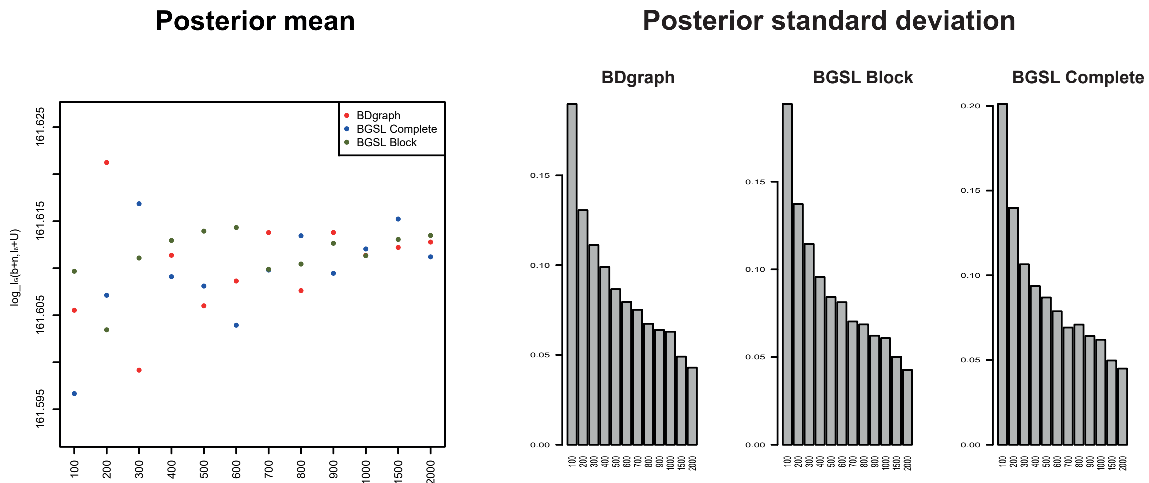

Figure 1.6. Monte Carlo empirical mean for posterior normalizing constant (left) and the corresponding standard deviation (right). The underlying graph has $p = 6$ nodes. Plots are given in logarithmic scale, obtain with 500 repetitions for each possible choice of Monte Carlo iterations.

As expected, the standard deviation decreases if we increase the number of iterations and, consequently, the empirical mean stabilizes.

Given the same graph, we also computed the same quantities for what concerns the posterior normalizing constant, see Figure 1.6. More details about this test are given in Chapter 4. Here we only want to underline that the empirical standard deviation for this second case still remains under control, even if it is higher than the previous example.
We can conclude that, in this low dimensional case, the approximations are very reliable, even if a limited number of iterations is performed, let us say about 500.
As reported in Figure 1.7, the scenario dramatically changes if we perform an high dimensional experiment, same test as before but on a 40 nodes graph. Several differences can be noticed

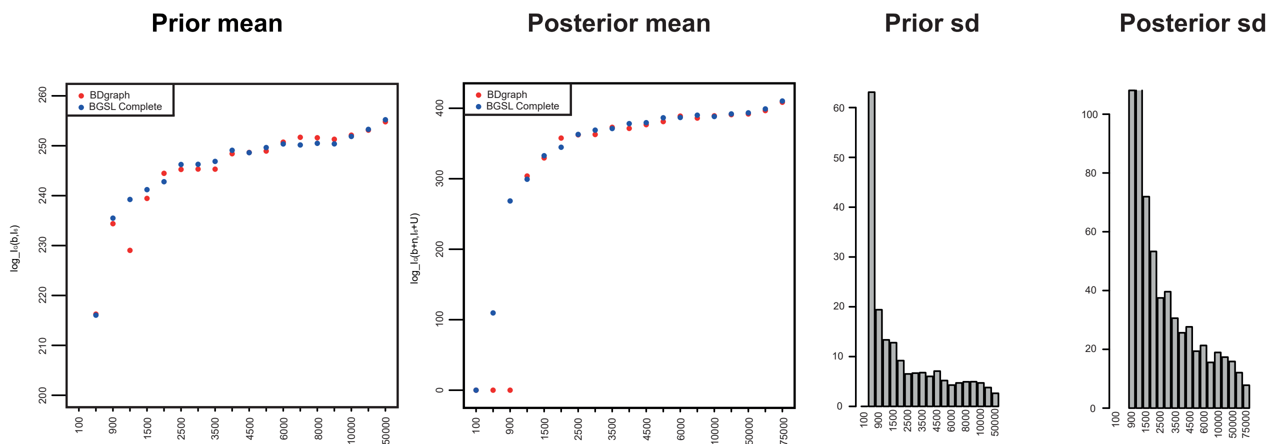

Figure 1.7. Monte Carlo empirical mean empirical standard deviation for prior and posterior normalizing constant. The underlying graph has $p = 40$ nodes. We only reported the standard deviation obtained with BGSL. Plots are given in logarithmic scale, they were obtained with 500 repetitions for each possible choice of Monte carlo iterations. If too little iterations are performed, the estimated value is below machine zero and therefore its logarithm is −infinity. Those cases are reported in the plots by setting the mean equal to zero and no bar in the standard deviation histogram.

with respect to the low dimensional example. First of all, we see that in this case a minimum number of iterations is required to get at least a numerical meaningful output. Indeed, because of the definition of (1.26), if not enough iterations are performed, the estimated value is below the machine zero and therefore its logarithm is −infinity. If the number of Monte Carlo iterations is sufficiently large, let us around 2500 the empirical standard deviation for the prior normalizing constant settles down 10, which still is still an high value if we take into account that the results are given in logarithmic scale. However, this problem gets much worst for what concerns the posterior case. The estimates we get by means of 10000 sampled values still have a standard deviation higher than 20.
Moreover, if this computation has to be done multiple times for each step of a MCMC method, so many iterations would not even be feasible. Finally, we also highlight that, in both cases,

the sample means do not even reach a stationary value.

Another important consideration about this topic regards the possibility of an approximation to the ratio of the prior normalizing constants. As explained in Section 1.3.2 the Bayes factor to compare models governed by G and G' is a function of the ratios of the prior and posterior normalizing constants. The method we propose in Chapter 2 is actually able to avoid both calculations. As later explained, the step needed to circumvent the prior ratio is rather complex and one may wonder if it is worth it. Figure 1.7 should be sufficiently convincing about the need not to compute any normalizing constant.
The question is if there is a simple way to at least approximate $I_G(b, D)/I_{G'}(b, D)$. A possible estimate was first verified numerically (Mohammadi and Wit 2012) and then proved (Letac et al. 2017).
It states that, given G, if we set D equal to $\boldsymbol{I}_p$ and $G' = G^{-e}$, that is the graph we obtain by removing the edge $e = (i, j)$, then the prior normalizing constant ratio can be approximated as

$$\frac{I_{G^{-e}}(b, \boldsymbol{I}_p)}{I_G(b, \boldsymbol{I}_p)} \approx \frac{1}{2\sqrt{\pi}} \frac{\Gamma\left(\frac{b+d}{2}\right)}{\Gamma\left(\frac{b+d+1}{2}\right)} \tag{1.29}$$

where d is the number of paths of length two linking the two end points i and j of e. The proof of (1.29) and estimates assessing its precision can be found in the forecited paper.
What we want to stress is that this estimate is valid as far as the two compared graphs differ only for a single link. What we want to achieve is instead a MCMC method such that we modify more that one link at a time, in principle we would like to change an arbitrary number of edges. We are not aware of any estimates for that case.

Summing up, in this section we presented the Monte Carlo method by Atay-Kayis and Massam (2005) that allows to estimate the GWishart normalizing constant also for non decomposable graphs. Thanks to a couple of experiments, we presented some limitations of such algorithm, which is reliable only in low dimensional but it struggle in the high dimensional one. A suggested rule of thumb may be $p < 10$. This issue is not new to the literature, it is further analyzed in Lenkoski and Dobra (2011), where the Laplace approximation method is proposed indeed to substitute the computation of the posterior normalizing constant. We did not implement that kind of approximation as it was not much considered in subsequent literature.

1.3.2 The marginal posterior probability of the graph

Drawing samples from the posterior of a Bayesian Gaussian graphical model, defined by (1.9), is a problem that has already been addressed in the literature. We now review some existing

approaches, keeping in mind that they all operate in the space $\mathcal{G}$ of all possible complete graphs. Our goal is instead to look for a block graph in $\mathcal{B}$ which, as we explain in Section 2.5, could in principle describe also the independence structure of unknown parameters, not just of the observed data.
We now start sketching the problem, providing also some examples of efficient methods that unfortunately are not suitable for the generalization we have in mind.

Given model (1.9) and recalling that $\boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n\}$, the joint density of $(\boldsymbol{Y}, \boldsymbol{K}, G)$ is equal to

$$f(\boldsymbol{Y}, \boldsymbol{K}, G) = \frac{|\boldsymbol{K}|^{(b+n-2)/2}\ \pi(G)}{(2\pi)^{np/2}\ I_G(b, D)} \exp\left\{-\frac{1}{2}\langle \boldsymbol{K}, D + U\rangle\right\} \tag{1.30}$$

where, for convenience, we denote the inner product $tr(AB^T)$ of symmetric $p \times p$ matrices A and B by $\langle A, B\rangle$. As already mentioned, the U matrix that appears in (1.30) is defined as $U = \sum_{i=1}^{n} \boldsymbol{y}_i \boldsymbol{y}_i^T$.
Since the GWishart distribution is conjugate to the likelihood, see (1.8), the precision matrix can be integrated out, leading to the following marginalized likelihood

$$P(\boldsymbol{Y} \mid G) = \frac{1}{(2\pi)^{np/2}} \frac{I_G(b+n, D+U)}{I_G(b, D)} \tag{1.31}$$

and therefore the posterior density of G is

$$P(G \mid \boldsymbol{Y}) \propto \pi(G)\ \frac{J_G(b, n, D, U)}{\sum_{G' \in \mathcal{G}} J_{G'}(b, n, D, U)} \tag{1.32}$$

where

$$J_G(b, n, D, U) = \frac{I_G(b+n, D+U)}{I_G(b, D)} \tag{1.33}$$

The problem can be reformulated as a model average estimate of the precision matrix $\boldsymbol{K}$ based on using the posterior probabilities given by (1.32) of each graph considered. We refer to

$$P^*(G \mid \boldsymbol{Y}) = \pi(G) J_G(b, n, D, U) \tag{1.34}$$

as the unnormalized graph score (Lenkoski and Dobra 2011).
As explained at the beginning of this section, the entire space of graphical models can not be enumerated and stochastic search methods have to be applied. Most of the works in this area focused on MCMC techniques that are desirable as they theoretically allow one to sample form the posterior model probability space, given a suitable burn-in period to reach convergence. Our proposed approach is based on MCMC model search and it is carefully explained in the next chapter.
We now want to present a different technique, which, even if it has less powerful theoretical

properties, is in practice competitive with a Monte Carlo Markov chain method. However, it is not directly applicable to our model.
Jones et al. (2005) noticed that, as p gets larger, MCMC methods require a considerable number of iterations to achieve convergence. Moreover, the Markov chain is merely a device to explore the graph space, since the posterior probability of each visited graph is readily available up to the normalizing constant,

$$\frac{1}{\sum_{G' \in \mathscr{G}} J_{G'}(b, n, D, U)} \tag{1.35}$$

see (1.32). Therefore there is no explicit need to repeatedly revisit G, it is enough to compute its score (1.34) once.
That is the key idea behind the *shotgun stochastic search* (SSS) method, which can be summarized by the steps collected in Algorithm 2.

Algorithm 2: Shotgun Stochastic Search

1. Start from graph G.
2. Select a number n_1 of graphs G' in the one edge away neighbourhood of G and compute their scores using (1.34).
3. Among the top $n_2 < n_1$ higher scores computed in step 2, choose one of the visited graphs proportionally to their score.
4. Return to step 2 and iterate. Keep a list of the overall best n_3 visited graphs.
5. Normalize the scores of the visited graphs by approximate (1.35) only within the list of the n_3 ones. According to the authors who proposed the SSS, those results reflects the true posterior probabilities to the extend that they contain most of the posterior mass.

Successively, Scott and Carvalho (2008) discussed the concept of global moves, in which the graph proposed as the starting point of step 2 in Algorithm 2 is not necessarily been visited in the search. This enables to move quickly to alternative areas of the graph space and to escape from low probability paths. Finally, the *Mode Oriented Stochastic Search* (MOSS) (Lenkoski and Dobra 2011), integrates the concept of global moves, which is better defined later on, within the framework of the SSS, improving the performance of the latter.

In our opinion, those two approaches are worth mentioning as they represent a simple and intuitive approach to the problem that is not strictly related to the MCMC procedure. They proved themselves a valid option in practice, they are suitable for parallelization, they rapidly converge and given the same number of compared graph, they usually provide sharper estimates.

This does not necessarily imply that they are better than MCMC; the lack of theoretical results about their convergence properties can not be ignored. Moreover, in this way we are forced to compute the graph score that is proportional to the ratio of a prior and a posterior GWishart normalizing constant and, as explained in Section 1.3.1, it is an extremely challenging task that can actually be avoided thanks to carefully studied Markov chain.
Another drawback, which is also the reason why we did not use them in our model, is that they are not flexible enough. Indeed, they strongly rely on the hypothesis that the graph $\boldsymbol{G}$ is used to define the dependence structure of observed data $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$. What if we use a graph to describe the structure of some unknown parameters, let us say zero mean $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n$ Gaussian parameters. Within a *Gibbs sampling* strategy, that is a MCMC method, the graph is sampled given all other parameters. In particular, at each iteration, we are free to think of the $\boldsymbol{\beta}$s as the "data" needed to estimate the graph. Therefore, we can inherit all the machinery developed for model (1.9), just need to substitute $\boldsymbol{y}_i$ with $\boldsymbol{\beta}_i$. This is true only for one iteration at a time, between subsequent iterations, all unknown quantities have to be sampled again. Intuitively, the "data" are changing at each step.
Unfortunately it would not be possible to apply a SSS, the marginal posterior $P(\boldsymbol{G} \mid \boldsymbol{Y})$ can not be derived. Since it is not a MCMC, it can not be even incorporated within a Gibbs sampler to exploit the analogy of the βs as data.

Even if we did not use in practice a SSS, it shows that it is possible to boost performances bypassing the usage of a MCMC, that are usually characterized by low convergence and low acceptance ratio. To overcome this second issue, Mohammadi and Wit (2015) addressed the problem by proposing a MCMC method that is not an acceptance-rejection sampler. It is the *Birth and Death approach* and it is presented in the next section.

1.3.3 Birth and Death approach

The most common MCMC methods in structural learning are acceptance-rejection sampler. However, the size and the structure of the graph space makes it difficult to construct an efficient proposal distribution. This usually implies a low acceptance ratio. To overcame this issue, Mohammadi and Wit (2015) proposed a *Birth and Death Monte Carlo Markov chain*, called BDMCMC for short. This approach significantly differs from a standard MCMC as it is a continuous time Markov process, which means that time between jumps to a higher dimension (births) or smaller one (deaths) are random variables defined by a specific rate. Contrary to acceptance-rejection samplers, in this setting all moves are accepted, making the BDMCMC extremely efficient. A possible implementation can be found in the `R` package `BDgraph`.

This method, as it is presented in this section, is not easily generalizable to be framed in a block graph scenario. Nevertheless, we believe that if we were able to reinterpret our proposed

model within a Birth and Death setting, we would boost its efficiency. To present how this chain is constructed, we first need to do a simple recap about the theory of spatial Birth and Death processes developed by Preston (1977).

Jump Processes

The usual Birth and Death process is a continuous time Markov chain having as state space the non negative integer, which means that the state of the process is the number n of individuals that are alive. At each step a transition to $n+1$ can occur, representing a birth, or to $n-1$ that is, of course, a death.
The process is usually assumed to be homogeneous, i.e. the transition probabilities does not depend on time and the transition rates just depend only on the number of people that are alive and not on their position. In a more realistic scenario, we would expect, for example, birth rates to decrease and death rates to increase because of overcrowding. In such situation the evolution of the process may be critically influenced by where the individuals are and not only on how many there are of them.

The work by Preston (1977) aimed to introduce a process which takes into account the position of the individuals, deriving a new one, called spatial Birth and Death, that is a particular jump process. First of all, we briefly recall what a jump process is.

Let $(\Omega, \mathscr{F})$ be a measurable space, let $\lambda : \Omega \to \mathbb{R}^+$ be $\mathscr{F}-$measurable and $T : \Omega \times \mathscr{F} \to \mathbb{R}^+$ be a probability kernel. A stochastic process with state space $(\Omega, \mathscr{F})$ is called a **jump process** with **intensity** λ and **transition kernel** T if, given that the process is in the state $x \in \Omega$, then the waiting time to the next jump has exponential law with mean $1/\lambda(x)$, it is independent of past history and the probability that the following jump leads to a point in $F \in \mathscr{F}$ is equal to $T(x, F)$.
To construct a spatial Birth and Death we need to define a particular jump process, which means that we need to specify a measurable space, the intensity and the probability kernel. We do not go through details of the construction of the state space because it will be naturally defined as soon as we introduce the graphical Birth and Death chain. We denote with $(\Omega_n, \mathscr{F}_n)$ the measurable space describing the configuration of the whole process when there are n individuals alive and with $(\Omega, \mathscr{F})$ the union of all the possible configurations.
Let now $\beta, \delta : \Omega \to \mathbb{R}^+$ be $\mathscr{F}$-measurable functions, respectively called birth and death rates of the process. For what concern the intensity λ and the probability kernel T, they are defined as functions those rates.
In particular the waiting time is the minimum between the waiting time for a birth or a death, which implies that $\lambda(x) = \beta(x) + \delta(x)$.
Regarding the kernel, if the process is in state $x \in \Omega_n$ and a birth occurs, then the position where the process jumps to is given by the probability measure $T_\beta^{(n)}(x, \cdot)$ on $(\Omega_{n+1}, \mathscr{F}_{n+1})$,

where $T_\beta^{(n)}$ is called birth kernel and it is defined as a function of β. Analogously the death kernel is defined as $T_\delta^{(n)}$.
Finally we are ready to define the kernel probability of the jump process, $T : \Omega \times \mathscr{F} \to \mathbb{R}^+$, as a weighted sum of the birth kernel and the death kernel,

$$T(x, F) = \frac{\beta(x)}{\lambda(x)} T_\beta^{(n)}(x, F_{n+1}) + \frac{\delta(x)}{\lambda(x)} T_\delta^{(n)}(x, F_{n-1}) \tag{1.36}$$

This jump process generalizes a homogeneous Birth and Death chain. It is clear that the rates play a key role in the construction of the process. Of course, it is necessary to find some conditions on the rates that ensure that such a process actually exists, all the details can be found in Preston (1977, section 5-6).

Birth and Death MCMC Method

We now present how Mohammadi and Wit (2015) used the theory of the previous section to build a BDMCMC for Gaussian graphical model selection.
$\Omega = \bigcup_{\substack{G \in \mathscr{G} \\ \boldsymbol{K} \in \mathbb{P}_G}} (G, K)$ is chosen as state space, which means that the state of the process is the joint space of the graph and the precision matrix.
We recall that, given an undirected graph $G = (V, E)$, we denote by E the set of its edges, by $\mathscr{W}$ the set of all the possible ones and by $\bar{E} = \mathscr{W} \setminus E$ the absent links. See the beginning of Chapter 1 for their precise definitions.

The process explores the graph space by adding or removing an edge. Births and deaths are characterized in the following way:

Death: Each edge $e \in E$ dies independently of the others as a Poisson process with rate $\delta_e(K)$. Denote the overall death rate with $\delta(K) = \sum_{e \in E} \delta_e(K)$. When the death of an edge $e = (i, j)$ occurs, then the process jumps to a new state $(G^{-e}, \boldsymbol{K}^{-e})$, where $G^{-e} = (V, E \setminus \{e\})$ and $\boldsymbol{K}^{-e} \in \mathbb{P}_{G^{-e}}$ and we also assume that it differs from the previous state only for the entries in position $\{(i, j), (i, i), (j, i)\}$.
As in the previous section, the death transition kernel is defined in terms of the death rates. For each $e \in E$, $T_{\delta_e}^G(K, \cdot)$ denotes the probability that the process jumps from (G, K) to a point in the new state $\bigcup_{\boldsymbol{K}^* \in \mathbb{P}_{G^{-e}}} (G^{-e}, \boldsymbol{K}^*)$.
Hence, given $F \subset \mathbb{P}_{G^{-e}}$, the kernel is simply defined as

$$\begin{aligned} T_{\delta_e}^G(K, F) &= \sum_{\eta \in E : \boldsymbol{K} \setminus k_\eta \in F} \frac{\delta_\eta(\boldsymbol{K})}{\delta(\boldsymbol{K})} \\ &= \frac{\delta_e(\boldsymbol{K})}{\delta(\boldsymbol{K})} \mathbb{1}(\boldsymbol{K}^{-e} \in F) \end{aligned} \tag{1.37}$$

Birth: A new edge $e \in \bar{E}$ is born independently of the others as a Poisson process with

birth rate $\beta_e(\boldsymbol{K})$. As before, the overall birth rate is $\beta(\boldsymbol{K}) = \sum_{e\in\bar{E}} \beta_e(\boldsymbol{K})$. If the birth of an edge $e = (i,j) \in \bar{E}$ occurs, then the process jumps to a new state $(G^{+e}, \boldsymbol{K}^{+e})$, where $G^{+e} = (V, E \cup \{e\})$ and $\boldsymbol{K}^{+e} \in \mathbb{P}_{G^{+e}}$.
As for the death rates, we assume that the new matrix differs from the previous state only for the entries in position $\{(i,j),(i,i),(j,i)\}$. This time the birth transition kernel is more involved because of the birth of a new element that has to be properly drawn in such a way that the next matrix will still be positive semidefinite. That is the reason why, given $F \subset \mathbb{P}_{G^{+e}}$, for each $e \in \bar{E}$, $T^G_{\beta_e}(\boldsymbol{K}, \cdot)$ is defined as

$$T^G_{\beta_e}(\boldsymbol{K}, F) = \frac{\beta_e(\boldsymbol{K})}{\beta(\boldsymbol{K})} \int_{\boldsymbol{K}_e : \boldsymbol{K} \cup \boldsymbol{K}_e \in F} b_e(\boldsymbol{K}_e, \boldsymbol{K}) d\boldsymbol{K}_e \tag{1.38}$$

where $b_e(\boldsymbol{K}_e, \boldsymbol{K})$ is the proposed density for the elements to be updated and it is chosen later. Finally, Birth and Death processes are supposed independent. Hence the intensity of the jump process is $\lambda(\boldsymbol{K}) = \beta(\boldsymbol{K}) + \delta(\boldsymbol{K})$. We recall that this implies that the time between two consecutive jumps is exponentially distributed with mean equal to $1/\lambda(\boldsymbol{K})$.

Now that the skeleton of the chain is set, one has to properly choose the rates such that the invariant distribution is exactly the posterior distribution of the graph and the precision matrix, $P(G, \boldsymbol{K}|\boldsymbol{Y})$.
Preston (1977, section 8) states that a necessary and sufficient condition for a generic law μ on $(\Omega, \mathscr{F})$ to be the unique invariant law of a jump process is

$$\int_F \lambda(x) d\mu(x) = \int T(x, F) \lambda(x) d\mu(x), \quad \forall F \in \mathscr{F} \tag{1.39}$$

Hence, applying this result to the BDMCMC we get that $P(G, \boldsymbol{K}|\boldsymbol{Y})$ is invariant if, given $F \in \mathbb{P}_G$

$$\int_F \delta(\boldsymbol{K}) dP(\boldsymbol{K}, G|\boldsymbol{Y}) = \sum_{e\in E} \int_{\mathbb{P}_{G^{-e}}} \beta(\boldsymbol{K}^{-e}) T^G_{\beta_e}(\boldsymbol{K}^{-e}, F) dP(\boldsymbol{K}^{-e}, G^{-e}|\boldsymbol{Y}) \tag{1.40}$$

$$\int_F \beta(\boldsymbol{K}) dP(\boldsymbol{K}, G|\boldsymbol{Y}) = \sum_{e\in \bar{E}} \int_{\mathbb{P}_{G^{+e}}} \delta(\boldsymbol{K}^{+e}) T^G_{\delta_e}(\boldsymbol{K}^{+e}, F) dP(\boldsymbol{K}^{+e}, G^{+e}|\boldsymbol{Y}) \tag{1.41}$$

Those equations can be interpreted as balance conditions. They states that leaving the set $F \in \mathbb{P}_G$ due to a birth or a death must be balanced by a death from $\mathbb{P}_{G^{+e}}$ to F or a birth from $\mathbb{P}_{G^{-e}}$ to F.
A sufficient condition that guarantees those conditions to be true is choosing the proposal density as

$$b_e(\boldsymbol{K}_e, \boldsymbol{K}^{-e}) = \frac{P\left(\ (k_{ij}, k_{jj}) \mid \boldsymbol{K} \backslash (k_{ij}, k_{jj}), G, \boldsymbol{Y}\ \right)}{P\left(\ k_{jj} \mid \boldsymbol{K}^{-e} \backslash k_{jj}, G^{-e}, \boldsymbol{Y}\ \right)} \tag{1.42}$$

and by imposing the birth and death rates to satisfy the following equation, given by theorem 3.1 in (Mohammadi and Wit 2015, Theorem 3.1),

$$\delta_e(\boldsymbol{K})P(G,\boldsymbol{K}\backslash(k_{ij},k_{jj})|\boldsymbol{Y}) = \beta_e(\boldsymbol{K}^{-e})P(G^{-e},\boldsymbol{K}^{-e}\backslash k_{jj}|\boldsymbol{Y}), \quad \forall e \in \mathscr{W} \tag{1.43}$$

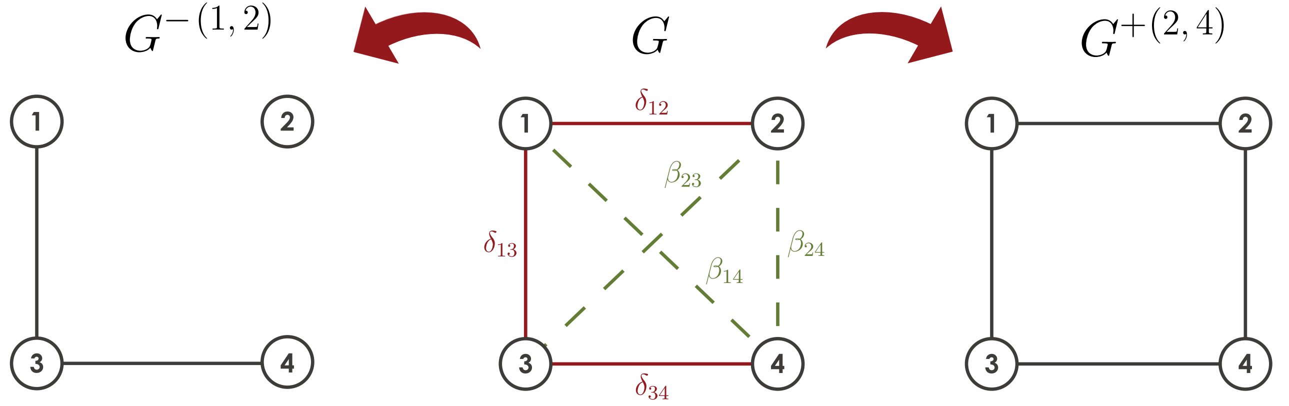

Figure 1.8. A visual representation of the Birth and Death chain. Current links may die proportionally to their death rates, δ, and the absent ones may be born proportionally to the βs.

Computing birth and death rates

The choice done by the authors is to choose the rates as

$$\beta_e(\boldsymbol{K}) = \min\left\{1, \frac{P(G^{+e},\boldsymbol{K}^{+e}\backslash(k_{ij},k_{jj}) \mid \boldsymbol{Y})}{P(G,\boldsymbol{K}\backslash k_{jj} \mid \boldsymbol{Y})}\right\}, \qquad \forall e \in \bar{E} \tag{1.44}$$

$$\delta_e(\boldsymbol{K}) = \min\left\{1, \frac{P(G^{-e},\boldsymbol{K}^{-e}\backslash k_{jj} \mid \boldsymbol{Y})}{P(G,\boldsymbol{K}\backslash(k_{ij},k_{jj}) \mid \boldsymbol{Y})}\right\}, \qquad \forall e \in E \tag{1.45}$$

it is easy to check that this particular choice actually satisfies (1.43) but it is not very appealing to be implemented in practice. An exhaustive analysis on how to further develop calculations for the quantities involved in (1.44) and (1.45) would be long and difficult. We limit ourselves to give a sufficient amount of information to explain why this particular choice of the transition kernels in not suitable for our case.
Let us only focus on the death rates; for each $e = (i,j) \in E$, the numerator is

$$P(G^{-e},\boldsymbol{K}^{-e}\backslash k_{jj} \mid \boldsymbol{Y}) = \frac{P(G^{-e},\boldsymbol{K}^{-e} \mid \boldsymbol{Y})}{P(k_{jj} \mid \boldsymbol{K}^{-e}\backslash k_{jj}, G^{-e}, \boldsymbol{Y})} \tag{1.46}$$

The full conditional for k_{jj} in $\boldsymbol{K}^{-e}$ is, up to a constant, a one-dimensional Wishart distribution (Roverato 2002, Lemma 1)

$$\begin{aligned} &k_{jj} - c \mid \boldsymbol{K}^{-e} \backslash k_{jj}, \boldsymbol{Y}, G^{-e} \sim \text{Wishart}\left(b+n, D^*_{jj}\right) \\ &\text{where } c = \boldsymbol{K}_{j,V\backslash j} \left(\boldsymbol{K}_{V\backslash j, V\backslash j}\right)^{-1} \boldsymbol{K}_{V\backslash j, j} \end{aligned} \tag{1.47}$$

where $\boldsymbol{D}^* = \boldsymbol{D} + \boldsymbol{U}$. For what concerns the numerator of (1.46), we only indicate Wang and Li (2012) for useful information. Result (1.47) can be generalized to a complete submatrix, let us say $\boldsymbol{K}_{\mathrm{I,I}}$ where I is a complete subset of indices. Such generalization is used to do analogous calculation in the denominator of (1.48), as the indices involved represents a simple 2×2 matrix placed on the diagonal. A possible extension to a situation where we would like to modify not just a single link but a whole block, possible not close to the diagonal, is not available.

After all computation, (1.44) and (1.45) can be rewritten as

$$\delta_e(\boldsymbol{K}) = \min\left\{1, \frac{P(G^{-e})}{P(G)} \frac{\boldsymbol{I}_G(b, D)}{\boldsymbol{I}_{G^{-e}}(b, D)} H(\boldsymbol{K}, D^*, e)\right\} \tag{1.48}$$

where

$$\begin{aligned} H(K, D^*, e) &= \left(\frac{D^*_{jj}}{2\pi(k_{ii} - k^1_{ii})}\right)^{\frac{1}{2}} \\ &\times \exp\left\{-\frac{1}{2}\left[\langle D^*, \boldsymbol{K}^0 - \boldsymbol{K}^1\rangle - \left(D^*_{ii} - \frac{D^*_{ij}}{D^*_{jj}}\right)\left(k_{ii} - k^1_{ii}\right)\right]\right\} \end{aligned}$$

that is a function of known quantities, indeed $\boldsymbol{K}^0 = \boldsymbol{K}$ expect from a null value in the positions (i, j) and (j, i) and an entry equal to c, defined in (1.47) in the diagonal position (j, j). $\boldsymbol{K}^1 = \boldsymbol{K}$ but entries equal to $\boldsymbol{K}_{e,V\backslash e} \left(\boldsymbol{K}_{V\backslash e, V\backslash e}\right)^{-1} \boldsymbol{K}_{V\backslash e, e}$ in the four positions corresponding to $\boldsymbol{e} = (i, j)$. Therefore, it is easy and fast to compute. For what concern the birth rates, the procedure and the resulting expression are almost identical and we do not report them here, see (Mohammadi and Wit 2015, section 3) for more details.

As anticipated in Section 1.3.1, the comparison of graphical models always has to face the issue of dealing with the ratio of the prior normalizing constants, as in (1.48). In this case, it is particularly annoying as it is involved in the definition of every birth and death rate and it is mandatory to avoid its Monte Carlo approximation. As the chain modifies only one link at a time, it is possible to exploit the approximation given in (1.29), which may result a bit rough, but it is in practice very helpful, allowing for a very fast and cheap approximation. It is not the only choice, another option makes use of the *Exchange algorithm* (Murray et al. 2012), that is later presented since it is a fundamental component of our proposed method.

The Birth and Death approach to Gaussian graphical models is resumed in Algorithm 3.

Algorithm 3: Birth and Death chain for GGM

Suppose the chain to be in the state $(G^{[s]}, \boldsymbol{K}^{[s]})$, iterate the following steps:

Step 1. Birth and Death step

- **1.1** $\forall e \in \bar{E}$, compute $\beta_e(\boldsymbol{K})$ and denote with $\beta(\boldsymbol{K})$ their sum.
- **1.2** $\forall e \in E$, compute $\delta_e(\boldsymbol{K})$ as in (1.48) and denote with $\delta(\boldsymbol{K})$ their sum.
- **1.3** Compute the expected holding time $w(\boldsymbol{K}) = 1/(\beta(\boldsymbol{K}) + \delta(\boldsymbol{K}))$.
- **1.4** Simulate the type of jump proportionally to the computed rates. Denote by $G^{[s+1]}$ the new graph.

Step 2. Sample a new precision

$$\boldsymbol{K}^{[s+1]} \mid G^{[s+1]} \sim \text{GWishart}(b + n, D + U) \tag{1.49}$$

Step 2 can be accomplished thanks to the exact sampler described in Lenkoski (2013), and further details are given in Section 3.3. We recall that the time between two consecutive jumps is exponentially distributed with mean equal to $w(\boldsymbol{K})$, also called weight of the state, which plays a key role in the a posterior selection of the graph.

We here conclude the overview about the Gaussian graphical models. We illustrated the problem and presented some efficient alternative for acceptance-rejection MCMC methods, which, according to the literature, may lead to better performance. However, the SSS approach is not suitable for us since it is not sufficiently general, while the BDMCMC would be a very smart choice, but unfortunately, the particular chain developed in Mohammadi and Wit (2015) can not be generalized to a scenario involving more that a single link modification. This does not imply that the Birth and Death paradigm can not be applied at all, the problem is this specific definition of the transition kernels, we would need to propose some new ones. For sure this is not an easy task, but in our opinion it is an option worth being analyzed more in depth.

Chapter 2

Block Double Reversible Jumps for Gaussian Graphical Models

In Chapter 1 we presented the problem and gave an overview about some of the existing methods for sampling from the posterior of model (1.9). This chapter is instead dedicated to illustrate our proposed approach. We recall that our goal is to derive a model that forces the estimated graph to be in the space of block structured graph $\mathscr{B}$. In practice, limiting our search to this particular subspace will improve its efficiency as the space to be inspected is smaller than the one of all possible graphs $\mathscr{G}$.

First of all we need to introduce some simple notations. Given $G \in \mathscr{G}$, we define **local move** a step of the Markov chain that leads to a new graph $G' = (V, E')$ that belongs to the *one edge away* neighbourhood of G, that are all those graphs such that $| \, |E| - |E'| \, | = 1$. Subsequent local moves define a **path** within the graph space $\mathscr{G}$ of visited graphs that are all neighbours one another. One of the implication of dealing with such a large space is that the chain may get stacked in path made of graphs with low posterior probability, simply called low probability path. Since only one edge at a time is modified, it may take many iterations to escape from those situations. That is why it would be preferable to combine local and **global moves**, that instead makes a jump to a new probability region of the graph space. Designing global steps is more challenging than the local counterpart because they need to preserve the convergence of the chain. Intuitively, if we reached convergence and we would perform a move by chance, we could end up in a lower probability region and it would take may time to recover convergence (Scott and Carvalho 2008).

Our procedure is innovative because we propose moves that modifies not only a single link but a whole block of them which, in principle, may be of arbitrary size.
In other words, our moves are local in $\mathscr{B}$ but not in $\mathscr{G}$. We are not aware of any existing method which modifies more than an element at a time, not even a SSS or a BDMCMC, see Section 1.3. Some authors do successfully combined local and global moves (Scott and

Carvalho 2008; Zhu et al. 2016) but also in this case moves are not done changing more than one link. What we are proposing is still different, it could even be bettered by taking into account global moves in $\mathscr{B}$. Experience did not evidence this needs, indeed our steps already modifies many edges.

We named our method **Block Double Reversible Jumps** for Gaussian graphical model . It is a trans-dimensional Monte Carlo Markov chain approach and it is scalable since it avoids any normalizing constant calculation.
This chapter describes all needed step to properly introduce it. First, we present a simple and naive MCMC called **Add-Remove Metropolis Hastings** for Gaussian graphical model and we show how to generalize it to perform a block graph search. It is an intuitive procedure and even if in practice it is not efficient, it is the starting point for more complex chains that are proposed in the literature. Among them, we had to identify the ones suitable for the extension we had in mind. For example, the method described in Wang and Li (2012), which is also the foundation of the Birth and Death chain, is not.
We recognized the needed potential in the **Reversible Jumps** chain for GGM, (Dobra et al. 2011), which relies on the properties of the Cholesky decomposition of a GWishart distributed precision matrix. As the name suggests, it is a *Reversible jumps* chain, whose general framework is briefly introduced in Section 2.2.1.
The resulting procedure is not yet fully scalable because of the presence of the ratio of prior normalizing constants. The *Exchange algorithm* comes in our aid, it combines with the Reversible Jumps chain to generate the **Double Reversible Jumps** method for Gaussian graphical model, whose generalization to block graphs is finally our novel proposal.

2.1 MCMC methods for Gaussian graphical models

Consider model (1.9), that is also reported below

$$\begin{aligned} \boldsymbol{y}_1, \ldots, \boldsymbol{y}_n \mid \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\mathbf{0},\ \boldsymbol{K}\right) \\ \boldsymbol{K} \mid G &\sim \text{GWishart}\left(b, D\right) \\ G &\sim \pi\left(G\right) \end{aligned}$$

where $\boldsymbol{y_i} \in \mathbb{R}^p$, $\forall i = 1 : p$ and $\boldsymbol{K}$ is a precision matrix belonging to $\mathbb{P}_G$, that is the space of all symmetric and positive definite matrices constrained by the graph $\boldsymbol{G}$. As in the previous chapters, we shorten the notation by defining $\boldsymbol{Y} = \{\boldsymbol{y_1}, ..., \boldsymbol{y}_n\}$.
The simplest possible procedure for sampling from its posterior distribution is to build a chain that first perform a *Metropolis-Hastings* step for sampling $\boldsymbol{G}$ from its marginal posterior distribution and then it extracts $\boldsymbol{K}$ from its full conditional.

Since we chose the GWishart prior because of its conjugacy property, see (1.8), the only difficulty in performing the second step consists just in being able to draw samples from this distributions, whose density is reported in (1.6). An efficient and direct sampler was proposed by Lenkoski (2013), it is carefully presented in Section 3.3.1.
The first step, drawing G from its marginal posterior law, is also apparently harmless. Such distribution has been introduce in Section 1.3.2 and it is proportional to the ratio of prior and posterior GWishart normalizing constants,

$$P(G \mid \boldsymbol{Y}) \ \propto \ \pi(G)\frac{\boldsymbol{I}_G(d+n, D+U)}{\boldsymbol{I}_G(d, D)} \tag{2.1}$$

where b and D are the prior hyperparameters and $U = \sum_{i=1}^{n} \boldsymbol{y}_i \boldsymbol{y}_i^T$.

To get a sample from (2.1), we can perform a *Metropolis-Hasting* step by simply proposing the addition or removal of one edge. This is call **Add-Remove Metropolis Hastings**, ARMH for short. See, among others, Giudici and Castelo (2003); Bhadra and Mallick (2013). The procedure is simple and intuitive, let us call $G^{[s]}$ the current graph in the chain, then we define by $nbd_p^+(G)$ the set of graphs having p nodes that can be obtained by adding an edge to $G \in \mathcal{G}$. Analogously, $nbd_p^-(G)$ is the set of graphs obtainable by removing an edge. We call $nbd_p(G) = nbd_p^+(G) \cup nbd_p^-(G)$ the one edge way neighborhood of G, that is already been cited at the beginning of the chapter.
ARMH is composed as follow;

1. Propose a new graph $G' \in nbd_p(G^{[s]}) \subset \mathcal{G}$ by either

   (a) removing an existing link of G. This happens with probability $(1 - \alpha_G)$.

   (b) adding a new link to $G^{[s]}$. This happens with probability α_G.

   Once that the proposed type of jump is decided, choose which link has to be added or removed with uniform probability among the possible ones.
   This procedure is equivalent to propose the new graph from the following proposal distribution,

$$q(G'|G^{[s]}) = \alpha_G \text{Unif}(nbd_p^+(G^{[s]})) + (1 - \alpha_G)\text{Unif}(nbd_p^-(G^{[s]})) \tag{2.2}$$

2. Set the proposal quotient equal to

$$\frac{q(G^{[s]}|G')}{q(G'|G^{[s]})} = \begin{cases} \frac{(1-\alpha_G)}{\alpha_G} \frac{|\ nbd_p^-(G^{[s]})\ |}{|\ nbd_p^+(G')\ |}, & \text{if removing an edge} \\ \\ \frac{\alpha_G}{(1-\alpha_G)} \frac{|\ nbd_p^+(G^{[s]})\ |}{|\ nbd_p^-(G')\ |}, & \text{if adding an edge} \end{cases} \tag{2.3}$$

Remark: if $\alpha_G = 0.5$, then the first quotient is equal to 1 in both cases. To lighten the notation, we always refer to this case.
In this case, (2.2) gives an equal probability of proposing to delete an edge from the current graph and of proposing to add one. We prefer this proposal with respect to the simpler case Unif($nbd_p(G^{[s]})$) (Madigan et al. 1995). Indeed, when sampling from that distribution, the probability of proposing a move that adds (or deletes) and edge from $G^{[s]}$ would be extremely small if the current graph has a very large (or small) number of edges, leading to poor mixing in the resulting Markov chain (Dobra et al. 2011).

3. Compute $P(G^{[s]} \mid \boldsymbol{Y})$ and $P(G' \mid \boldsymbol{Y})$ from equation (2.1).

4. Suppose a link has to be added, jump from $G^{[s]}$ to G' with probability

$$\begin{aligned}\gamma(G^{[s]} \to G') &= \min\left\{1, \frac{P(G' \mid \boldsymbol{Y})\; q(G^{[s]}|G')}{P(G^{[s]} \mid \boldsymbol{Y})\; q(G'|G^{[s]})}\right\} \\ &= \min\left\{1, \frac{\pi(G')}{\pi(G^{[s]})}\,\frac{\boldsymbol{I}_{G^{[s]}}(d,D)}{\boldsymbol{I}_{G'}(d,D)}\,\frac{\boldsymbol{I}_{G'}(d+n,D+U)}{\boldsymbol{I}_{G^{[s]}}(d+n,D+U)}\,\frac{|nbd_p^+(G^{[s]})|}{|nbd_p^-(G')|}\right\}\end{aligned} \tag{2.4}$$

Algorithm 4: Add-Remove Metropolis Hastings

Step 1. Given $G^{[s]} \in \mathscr{G}$, propose $G' \in nbd_p(G^{[s]}) \subset \mathscr{G}$ from (2.2). Suppose a link has to be added.

Step 2. Jump from $G^{[s]}$ to G' with probability given by (2.4)

Algorithm 4 summarizes the previously presented steps. As displayed in Figure 2.1, the model comparison probability γ does not explicitly involve the precision matrix. It is implicitly taken in consideration as (2.1) is obtain integrating out $\boldsymbol{K}$. This is a very simple procedure

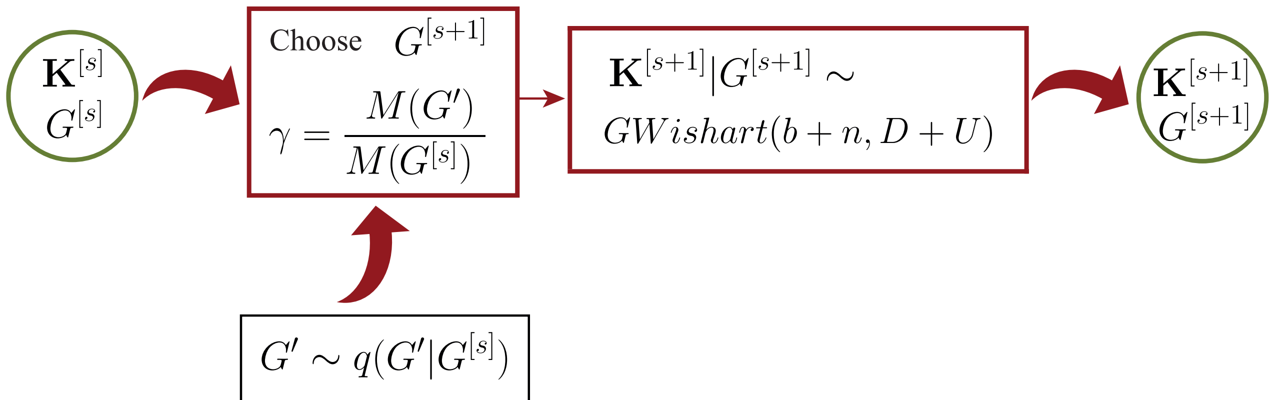

Figure 2.1. Visual representation of the ARMH.

that is also straightforward to be generalized so that we only visit graphs belonging to $\mathscr{B}$. The

only modification required is the way a new graph is proposed, i.e step 1 of in Algorithm 4. Suppose $G^{[s]} \in \mathscr{B}$, to propose a new graph $G' \in \mathscr{B}$ we can just map the current graph into its block form representative $G_A^{[s]} \in \mathscr{G}_A$. We now propose G'_A by adding or removing a link of $G_A^{[s]}$ and we map back the result to obtain G'. It consists drawing $G'_A \in nbd_M(G_A)$ from the proposal distribution

$$q(G'_A|G_A^{[s]}) = \frac{1}{2}\text{Unif}(nbd_M^+(G_A^{[s]})) + \frac{1}{2}\text{Unif}(nbd_M^-(G_A^{[s]})) \tag{2.5}$$

The candidate graph G' is then defined as $G' = \rho(G'_A)$. A visual representation in given in Figure 2.2.

$nbd_M(G_A^{[s]})$ simply denotes the one-edge away neighbourhood of $G_A^{[s]}$, whose definition is

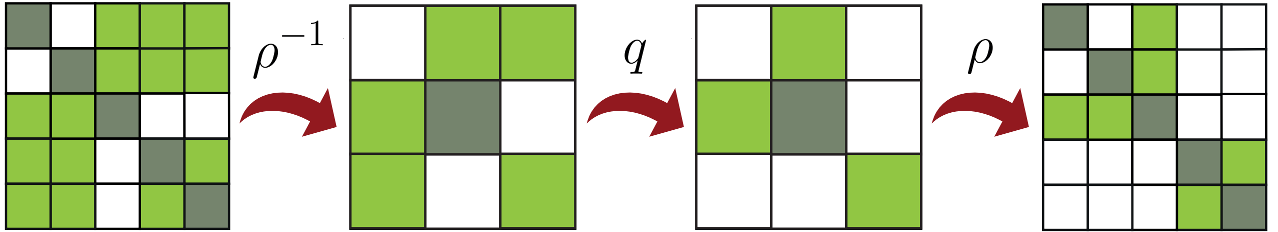

Figure 2.2. Visual representation of how a block graph is proposed. The leftmost panel is the current graph $G^{[s]}$ and the rightmost is the proposed one, G'.

analogous to $nbd_p(G^{[s]})$ but in the space of multigraphs $\mathscr{G}_A$, i.e all those configurations that can be reached by adding or removing a link between groups of variable. nbd_M^+ and nbd_M^- have the same meaning of nbd_p^- and nbd_p^- respectively.

The resulting method for block graphs only is called **Block Add-Remove Metropolis Hastings**, Block ARMH for short, and it is summarized in Algorithm 5,

Algorithm 5: Block Add-Remove Metropolis Hastings

Step 1. Given $G^{[s]} \in \mathscr{B}$, propose $G' \in nbd_M(G_A^{[s]}) \subset \mathscr{B}$ by:

1.1 Computing $G_A^{[s]} = \rho^{-1}(G^{[s]})$.

1.2 Proposing G'_A from (2.5).

1.3 Mapping the proposed block graph back to $\mathscr{B}$, $G' = \rho(G'_A)$.

Step 2. Suppose a block has to be added, jump from $G^{[s]}$ to G' with probability

$$\gamma(G^{[s]} \to G') = \min\left\{1, \frac{\pi(G')}{\pi(G^{[s]})}\frac{\boldsymbol{I}_{G^{[s]}}(d,D)}{\boldsymbol{I}_{G'}(d,D)}\frac{\boldsymbol{I}_{G'}(d+n,D+U)}{\boldsymbol{I}_{G^{[s]}}(d+n,D+U)}\frac{|nbd_M^+(G_A^{[s]})|}{|nbd_M^-(G'_A)|}\right\} \tag{2.6}$$

The simplicity characterizing Algorithm 4 and Algorithm 5 is very appealing but, unfortunately, they become useless as soon as the size of the graph grows. Both (2.4) and (2.6) requires

a graphical model comparison based on the ratio of both prior and posterior normalizing constants. In the previous chapter, see Section 1.3.1, we pointed out that those constants are intractable and, because of high variance problems, their Monte Carlo approximations are unreliable and unstable in high dimension. The presence of $I_G(b+n, D+U)$ is particularly annoying as it emphasizes such problem and no cheap approximation as (1.29) is available. The following step is then to understand how to get rid of that ratio.

2.2 Eliminating the posterior normalizing constant

We here present how to bypass the need of computing the ratio of GWishart posterior normalizing constants. That term arises integrating out the precision matrix from the joint posterior distribution. Thanks to the explicit formula for the marginal posterior of the graph, see (2.1), in Section 2.1 we derived a Metropolis Hasting step that does not involve explicitly $\boldsymbol{K}$, its information are implicitly taken into account by $I_G(b+n, D+U)$.

The only way to avoid dealing with that term is to build up a proposal step that explicitly considers both G and K, i.e on the joint space of graph and precision matrix. From a theoretical point of view, this choice imply the construction of chains that are more involved than the ones presented in Algorithm 4 and Algorithm 5. Difficulties are due to the fact that the structure of $\boldsymbol{K}$ is constrained by $\boldsymbol{G}$, indeed, we recall that $\boldsymbol{K} \in \mathbb{P}_G$.

Suppose the chain to be in state $(\boldsymbol{K}^{[s]}, \boldsymbol{G}^{[s]})$ and we are proposing $(\boldsymbol{K}', \boldsymbol{G}')$ by adding a new link $e = (i, j)$. Those two models are characterized by a different number of unknown parameters, as the first is constrained to set $k_{ij} = k_{ji} = 0$ but the second is not.
Note that Algorithm 4 and Algorithm 5 do not have to deal with this problem because $\boldsymbol{K}$ is marginalized out.

An extension to standard MCMC methodology that allows simulations of the posterior distribution on spaces of varying dimensions is called **Reversible Jumps Monte Carlo Markov chain**, or RJMCMC for short. Let us revise its fundamental ideas.

2.2.1 Overview of Reversible Jumps MCMC

The RJMCMC (Green 1995) provides a general framework for a MCMC simulation in which the dimension of the parameter space can change between iterates of the Markov chain. First of all, let us introduce some notation, borrowing the one used in Sisson and Fan (2011).
Suppose that for observed data, denoted by $\boldsymbol{Y}$, we have a countable collection of candidate models $M = \{M_1, M_2, ...\}$ indexed by a parameter k. Each model M_k has a n_k dimensional vector of unknown parameters $\theta_k \in \mathbb{R}^{n_k}$. We are interested in a setting where n_k may be different according to the current model. The joint posterior of (θ_k, k) given $\boldsymbol{Y}$ is of course the product of the likelihood and the joint prior $P(\theta_k, k) = p(\theta_k|k)p(k)$, where $p(\theta_k|k)$ and

$p(k)$ denotes density functions. Hence, we have

$$P(\theta_k, k \mid \boldsymbol{Y}) \propto f(\boldsymbol{Y}|\theta_k, k)p(\theta_k|k)p(k) \tag{2.7}$$

The RJMCMC uses the joint posterior distribution (2.7) as a target of a MCMC sampler over the state space $\boldsymbol{\Omega} = \bigcup_k(\{k\} \times \mathbb{R}^{n_k})$. The extension of the Metropolis-Hastings, MH for short, algorithm is theoretically challenging because of the fact that the dimension of the states of the chain, (θ_k, k) may vary over the state space. However the resulting algorithm is surprisingly simple to understand as it mimics the same steps of a standard MH.
Also in this setting, Markov chain transitions from a current state $\boldsymbol{\theta} = (\theta_k, k) \in A$ in model M_k are realised by first proposing a new state $\boldsymbol{\theta'} = (\theta'_{k'}, k') \in B$ in model $M_{k'}$ from a proposal distribution $q(\boldsymbol{\theta}|\boldsymbol{\theta'})$. A and B are Borel subsets of $\boldsymbol{\Omega}$. In order to construct a time-reversible Markov chain we need to enforce the detailed balance condition through the acceptance probability $\gamma(\boldsymbol{\theta}, \boldsymbol{\theta'})$. The balancing equation then becomes

$$\int_{(\boldsymbol{\theta},\boldsymbol{\theta'})\in A\times B} p(\boldsymbol{\theta}|\boldsymbol{Y})q(\boldsymbol{\theta}|\boldsymbol{\theta'})\gamma(\boldsymbol{\theta}, \boldsymbol{\theta'})d\boldsymbol{\theta}d\boldsymbol{\theta'} = \int_{(\boldsymbol{\theta},\boldsymbol{\theta'})\in A\times B} p(\boldsymbol{\theta'}|\boldsymbol{Y})q(\boldsymbol{\theta'}|\boldsymbol{\theta})\gamma(\boldsymbol{\theta'}, \boldsymbol{\theta})d\boldsymbol{\theta}d\boldsymbol{\theta'} \tag{2.8}$$

where $p(\boldsymbol{\theta}|\boldsymbol{Y})$ and $p(\boldsymbol{\theta'}|\boldsymbol{Y})$ are the posterior distribution with respect to model M_k and $M_{k'}$ respectively. As usual, if rejected, the chain remains in the current state. One way to ensure (2.8) is by choosing the acceptance probability as

$$\gamma(\boldsymbol{\theta} \to \boldsymbol{\theta'}) = \min\left\{1, \frac{p(\boldsymbol{\theta'}|\boldsymbol{Y})q(\boldsymbol{\theta'}|\boldsymbol{\theta})}{p(\boldsymbol{\theta}|\boldsymbol{Y})q(\boldsymbol{\theta}|\boldsymbol{\theta'})}\right\} \tag{2.9}$$

and $\gamma(\boldsymbol{\theta'} \to \boldsymbol{\theta})$ is similarly defined. It is straightforward to note that (2.9) is definitely analogous to the usual MH acceptance ratio.
The main question is now how to construct proposal moves between different models. In practice this is achieved by means of the concept of *dimension matching.*
Suppose, without loss of generality, that we wish to propose a state $\boldsymbol{\theta'}$ such that $n_{k'} > n_k$. In order to match dimensions between states, a random vector $\boldsymbol{u}$ of length $d_{k\to k'} = n_{k'} - n_k$ is drawn from a known density $q_{k\to k'}(\boldsymbol{u})$. The current state θ_k and the random vector $\boldsymbol{u}$ are then mapped through a one-to-one mapping function $g_{k\to k'} : \mathbb{R}^{n_k} \times \mathbb{R}^{d_k} \to \mathbb{R}^{n_{k'}}$ in such a way that the new state is then defined as $\theta'_{k'} = g_{k\to k'}(\theta_k, \boldsymbol{u})$.
Taking into account (2.8) and the proposal constructed above, (2.9) becomes

$$\gamma(\boldsymbol{\theta} \to \boldsymbol{\theta'}) = \min\left\{1, \frac{p(\boldsymbol{\theta'}|\boldsymbol{Y})q(k' \to k)}{p(\boldsymbol{\theta}|\boldsymbol{Y})q(k \to k')q_{k\to k'}(\boldsymbol{u})}\left|\frac{\partial g_{k\to k'}(\theta_k, \boldsymbol{u})}{\partial(\theta_k, \boldsymbol{u})}\right|\right\} \tag{2.10}$$

where $q(k \to k')$ denotes the probability of proposing a move from model M_k to model $M_{k'}$ and the final term is the Jacobian of the transformation mapping θ_k and $\boldsymbol{u}$ into $\theta_{k'}$. This term

arises because of change of variables given by $g_{k\to k'}$ required in the integral equation (2.8). The reverse move proposal, that is from model $M_{k'}$ to M_k is made deterministically and it is accepted with probability

$$\gamma(\boldsymbol{\theta'} \to \boldsymbol{\theta}) = \frac{1}{\gamma(\boldsymbol{\theta} \to \boldsymbol{\theta'})}$$

The RJMCMC is in practice very similar to a standard MH. It is commonly composed of two consecutive type of moves, the so called *within model moves* where a model k is fixed and it is possible to update the parameters θ_k according to any MCMC updating scheme and *between model moves* characterized by a simultaneous update of model indicator k and parameters θ_k according to a general reversible proposal having an acceptance probability given by (2.10).

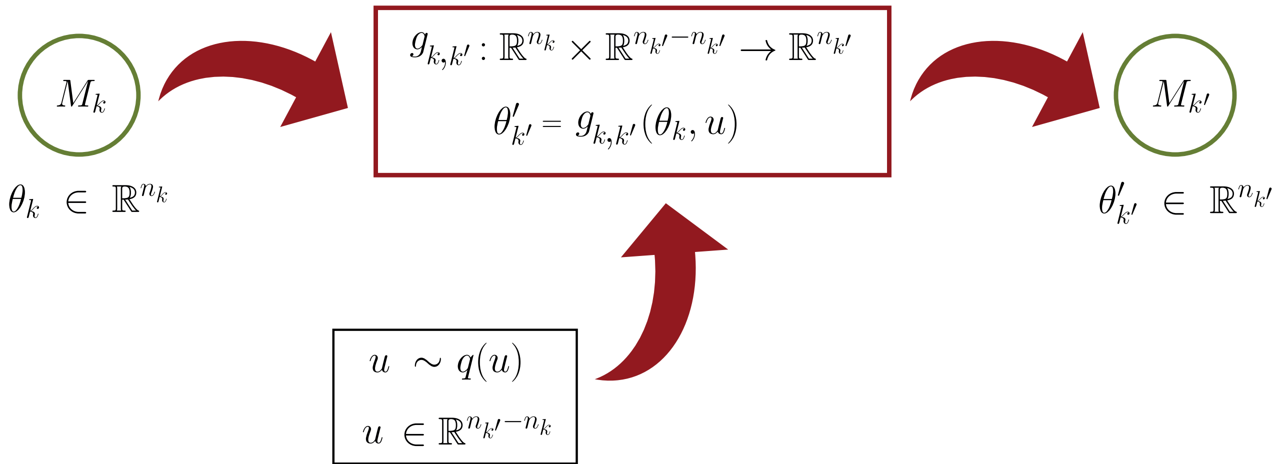

Figure 2.3. Visual representation of increasing dimension Reversible Jump from model M_k to M'_k, $n_{k'} > n_k$.

2.2.2 The Cholesky decomposition of `GWishart` random variates

The Reversible Jumps method we are going to present relies on perturbing elements of the Cholesky decomposition of the precision matrix $\boldsymbol{K}$. It is then needed a little review of some important results by Roverato (2002) and Atay-Kayis and Massam (2005).

Since $\boldsymbol{K} \in \mathbb{P}_G$, it always admits a Cholesky decomposition $\boldsymbol{K} = \Phi^T\Phi$ where $\Phi = (\phi_{ij})_{1\leq i\leq j\leq p}$ is upper triangular. The zero constraints on the off-diagonal elements of $\boldsymbol{K}$ imposed by $G = (V, E)$ induce a well-defined structure on Φ.

Denote with $\nu(G) = \{(i, j) : i = j \text{ and } i \in V \text{ or } (i, j) \in E\}$ the set of the diagonal elements and the links belonging to G. We call $\Phi^{\nu(G)} = \{\phi_{ij} \in \nu(G)\}$ set of **free elements** of Φ. The remaining entries, that we simply refer to as non-free elements, are uniquely determined through the completion operation (Atay-Kayis and Massam 2005, prop. 2) as a function of $\Phi^{\nu(G)}$. The reason of this decomposition is clarified in the following theorem

Theorem 2.2.2.1. *Let $\boldsymbol{K} \in \mathbb{P}_G$ and let $\boldsymbol{K} = \Phi^T\Phi$ be its Cholesky decomposition, where Φ is an upper triangular matrix. Then, the entries Φ_{ij} are such that,*

$$\phi_{ij} = \frac{k_{ij} - \sum_{h=1}^{i-1} \phi_{hi}\phi_{hj}}{\phi_{ii}} \quad \textit{if } (i,j) \in \nu(G) \tag{2.11}$$

For the case $(i,j) \notin \nu(G)$, i.e all those elements such that $k_{ij} = 0$, we have

$$\begin{aligned} \phi_{1j} &= 0 & \forall j = 2:p \\ \phi_{ij} &= \frac{- \sum_{h=1}^{i-1} \phi_{hi}\phi_{hj}}{\phi_{ii}} & 1 < i \le j \le p \end{aligned} \tag{2.12}$$

The proof is given in Roverato (2002) and Atay-Kayis and Massam (2005), it is simply based upon the existence and uniqueness of the Cholesky decomposition.
Looking at (2.11) it is clear why they are called free elements as they explicitly depends on the value of k_{ij} that is non null and completely arbitrary. Instead (2.12) shows that each entry ϕ_{ij} with $i < j$ and $(i,j) \notin E$ is a function of all the other elements ϕ_{rs} such that $(r,s) \overset{L}{<} (i,j)$, where $\overset{L}{<}$ symbol stands for lexicographic compare. This implies that they can be recursively obtained once the free elements are known.

Roverato (2002) proved that the Jacobian of the transformation mapping $\boldsymbol{K}$ into $\Phi^{\nu(G)}$ is

$$J(\boldsymbol{K} \rightarrow \Phi^{\nu(G)}) = 2^p \prod_{i=1}^{p} \phi_{ii}^{\nu_i^G + 1} \tag{2.13}$$

where

$$\nu_i^G = |\{j : j > i \text{ and } (i,j) \in E\}| \tag{2.14}$$

is the sum of elements in i-th row of adjacency matrix, from position $i+1$ up to the end. We also define $d_i^G = |\{j : j < i \text{ and } (i,j) \in E\}|$ that is the sum of elements in i-th row of adjacency matrix, from position 1 up to the $i-1$-th or, exploiting the symmetry of the matrix, the sum of elements in i-th column, from position 1 up to the $i-1$-th. It is clear that $\nu_i^G + d_i^G$ represents the number of neighbors of vertex i.
Finally we recall a well known property of the Cholesky decomposition that allows us to rapidly determine the determinant of $\boldsymbol{K}$. We have

$$\det(\boldsymbol{K}) = \prod_{i=1}^{p} \phi_{ii}^2 \tag{2.15}$$

Note that this formulation involves only diagonal values of Φ that are free elements by definition.

2.2.3 Stochastic search in the complete graph space

In this section we present a Reversible Jumps method for sampling from the joint posterior distribution of precision matrix $\boldsymbol{K} \in \mathbb{P}_G$ and graphs $\boldsymbol{G} \in \mathcal{G}$, thereby performing inference for both those parameters.

Consider model (1.9), we report it below for sake of clarity.

$$\begin{aligned} \boldsymbol{y}_1, \dots, \boldsymbol{y}_n \mid \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p(\boldsymbol{0}, \ \boldsymbol{K}) \\ \boldsymbol{K} \mid G &\sim \text{GWishart}(b, D) \\ G &\sim \pi(G) \end{aligned}$$

The goal is to sample from the joint posterior distribution

$$P(K, G|\boldsymbol{Y}) \propto P(\boldsymbol{Y}|K)P(K|G)\pi(G) \tag{2.16}$$

that is well defined as far as $\boldsymbol{K} \in \mathbb{P}_G$. A Reversible Jumps is needed as the addition or deletion of edges involves a change in the dimension of the parameter space since the corresponding element of $\boldsymbol{K}$ becomes free or constrained to zero.
Proposing a new precision matrix by modifying its elements is a dangerous operation since we may end up losing its positive definiteness. A safer procedure is instead to work on its Cholesky decomposition $\boldsymbol{\Phi}$. Thanks to the previously mentioned properties, after the change of variable $\boldsymbol{K} \mapsto \boldsymbol{\Phi}$, the unconstrained parameter are only the free elements $\boldsymbol{\Phi}^{\nu(G)}$ and a modification of one of those is reflected on the constrained values by the completion operation. For the moment we deal only with the case where just a single link at a time is modified, (Dobra et al. 2011), then we generalize this assumption in Section 2.2.4.

We denote the current state of the chain by $(\boldsymbol{K}^{[s]}, G^{[s]})$ with of course, $\boldsymbol{K}^{[s]} \in \mathbb{P}_{G^{[s]}}$. The next state $\boldsymbol{K}^{[s+1]} \in \mathbb{P}_{G^{[s+1]}}$ is reached by sequentially updating the edges of the graph given the current precision matrix and then the precision matrix given the current graph.

Step 1. Updating the graph

1.1. *Sampling G'*
Propose a new graph $G' \in nbd_p(G^{[s]}) \subset \mathcal{G}$ from (2.2). Assume $\alpha_G = 0.5$.
Without loss of generality we also assume G' to be obtained by adding the edge (i_0, j_0), $i_0 < j_0$ to $G^{[s]}$.
It is then straightforward to derive that $\nu(G') = \nu(G^{[s]}) \cup \{(i_0, j_0)\}$, $\nu_{i_0}^{G'} = \nu_{i_0}^{G^{[s]}} + 1$ and $\nu_{j_0}^{G'} = \nu_{j_0}^{G^{[s]}}$.

1.2. *Construction of* $\boldsymbol{K}'$

In order to define the precision matrix $\boldsymbol{K}'$ it is sufficient to assign all the free elements of $\boldsymbol{\Phi}'$. Indeed the remaining values can be derive with the completion operator, see Theorem 2.2.2.1. In particular we set $\phi'_{ij} = \phi^{[s]}_{ij}$ for all $(i, j) \in \nu(\boldsymbol{G}^{[s]})$. The only free element left is the one corresponding to the new link.
We sample

$$\eta \sim N(\phi^{[s]}_{i_0 j_0}, \sigma^2_g) \tag{2.17}$$

and set $\phi'_{i_0 j_0} = \eta$.
Note: Throughout this work, the σ^2_g parameter that appears in (2.17) denotes a variance, not a precision and it is a fixed parameters that would need to be tuned.
After having determined all the other values from $(\boldsymbol{\Phi}')^{\nu(G')}$ though the completion operation, we can finally define $\boldsymbol{K}' = (\boldsymbol{\Phi}')^T \boldsymbol{\Phi}'$.
We highlight that all the free elements of $\boldsymbol{\Phi}'$ and $\boldsymbol{\Phi}^{[s]}$ coincide, except of course the perturbed one in position (i_0, j_0).
In particular the diagonal entries do not change and this implies, by (2.15), that $\det(\boldsymbol{K}') = \det(\boldsymbol{K}^{[s]})$.

1.3. *Find the Jacobian of the Reversible Jump step*

As mentioned before, the dimensional of the parameter space increases by 1 as we jump form $(\boldsymbol{K}^{[s]}, \boldsymbol{G}^{[s]})$ to $(\boldsymbol{K}', \boldsymbol{G}')$. As explained in Section 2.2.1, when deriving the acceptance probability of the move, we would need to consider the Jacobian of the transformation mapping

$$(\ (\boldsymbol{\Phi}^{[s]})^{\nu(G^{[s]})},\ \eta\ ) \rightarrow (\boldsymbol{\Phi}')^{\nu(G')}$$

It is a map from $\mathbb{R}^{|\nu(G^{[s]})|} \times \mathbb{R}$ into $\mathbb{R}^{|\nu(G^{[s]})|+1}$. The procedure previously described is nothing more than a linear transformation,

$$(\boldsymbol{\Phi}')^{\nu(G')} = \boldsymbol{I}_{|\nu(G^{[s]})|+1}((\boldsymbol{\Phi}^{[s]})^{\nu(G^{[s]})},\ \eta)^T$$

that has of course Jacobian equal to 1.

1.4. *Acceptance probability*

In order to correctly derive the acceptance probability, whose general form is given in (2.10) with need to take into account the joint posterior of precision and graph, see (2.16), the proposal for the new graph $q(\boldsymbol{G}'|\boldsymbol{G}^{[s]})$, see (2.2) and the one for η. The Jacobian of the deterministic map is simply equal to 1.
We should not forget that, since we are working through $\boldsymbol{\Phi}$, we also have to consider

the two Jacobian $J(\boldsymbol{K}' \to (\Phi')^{\nu(G')})$ and $J(\boldsymbol{K}^{[s]} \to (\Phi^{[s]})^{\nu(G^{[s]})})$, whose form is stated in (2.13).

We finally derive that the probability for the move to be accepted is $\gamma\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \to (\boldsymbol{K}', G')\right) = \min\{1, R^{+}\}$, where R^{+} is given by

$$R^{+} = \sqrt{2\pi\sigma_g^2}\ \phi_{i_0 i_0}^{[s]}\ \frac{\boldsymbol{I}_{G^{[s]}}(b, D)}{\boldsymbol{I}_{G'}(b, D)}\ \frac{\pi(G')}{\pi(G^{[s]})}\ \frac{|nbd_p^{+}(G^{[s]})|}{|nbd_p^{-}(G')|} \times \\ \times\ \exp\left\{-\frac{1}{2}\left[\langle \boldsymbol{K}' - \boldsymbol{K}^{[s]},\ D + U\rangle - \frac{(\eta - \phi_{i_0 j_0}^{[s]})^2}{\sigma_g^2}\right]\right\} \tag{2.18}$$

We next assume that the candidate graph G' is obtained by deleting edge (i_0, j_0) from $G^{[s]}$. As explained in Section 2.2.1, we assign the acceptance probability as the inverse of the reverse, dimension increasing, move that we can compute thanks to (2.18).

$\gamma\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \to (\boldsymbol{K}', G')\right) = \min\{1, R^{-}\}$, where R^{-} is the inverse of the dimension increasing move. Writing it down explicitly we get

$$R^{-} = \left(\sqrt{2\pi\sigma_g^2}\ \phi_{i_0 i_0}^{[s]}\right)^{-1} \frac{\boldsymbol{I}_{G^{[s]}}(b, D)}{\boldsymbol{I}_{G'}(b, D)}\ \frac{\pi(G')}{\pi(G^{[s]})}\ \frac{|nbd_p^{-}(G^{[s]})|}{|nbd_p^{+}(G')|} \times \\ \times\ \exp\left\{-\frac{1}{2}\left[\langle \boldsymbol{K}' - \boldsymbol{K}^{[s]},\ D + U\rangle + \frac{(\phi'_{i_0 j_0} - \phi_{i_0 j_0}^{[s]})^2}{\sigma_g^2}\right]\right\} \tag{2.19}$$

We call $(\boldsymbol{K}^{[s+1/2]}, G^{[s+1]})$ the state of the chain at the end of step 1. By construction we have $\boldsymbol{K}^{[s+1/2]} \in P_{G^{[s+1]}}$.

Step 2. Updating the precision matrix

Given the updated graph $G^{[s+1]}$, we update the precision matrix by means of an exact sampler (Lenkoski 2013; Mohammadi and Wit 2015) for the GWishart distribution.

$$\boldsymbol{K}^{[s+1]} \mid G^{[s+1]} \sim \text{GWishart}(b + n, D + U) \tag{2.20}$$

We call this procedure **Reversible Jumps for GGM**, or simply RJ. A summarized description of this method is given in Algorithm 6 and Figure 2.4 provides a visual representation.

Algorithm 6: Reversible Jumps for GGM

Suppose the chain to be in state $(\boldsymbol{K}^{[s]}, \boldsymbol{G}^{[s]})$. For each iteration:

Step 1. Updating the graph

1.1 Sample $\boldsymbol{G}'$ from $q(G'|G^{[s]})$ given by (2.2). Suppose link (i_0, j_0) has to be added, see (2.19) for deleting moves.

1.2 Sample $\eta \sim N(\phi^{[s]}_{i_0 j_0}, \sigma^2_g)$

1.3 Set $(\Phi')^{\nu(G^{[s]})} = (\Phi^{[s]})^{\nu(G^{[s]})}$ and $\phi'_{i_0 j_0} = \eta$.
Derive the remaining elements by completion operation, see Theorem 2.2.2.1, and define $K' = (\Phi')^T \Phi'$.

1.4 Compute $\gamma\left(\left(\boldsymbol{K}^{[s]}, \boldsymbol{G}^{[s]}\right) \to (\boldsymbol{K}', \boldsymbol{G}')\right)$ using (2.18).
Draw $c \sim \text{Unif}[0,1]$.

1.5 **if** $c < \gamma$ **then** set $G^{[s+1]} = G'$.

Step 2. Updating the precision matrix:
Draw $\boldsymbol{K}^{[s+1]}$ from (2.20).

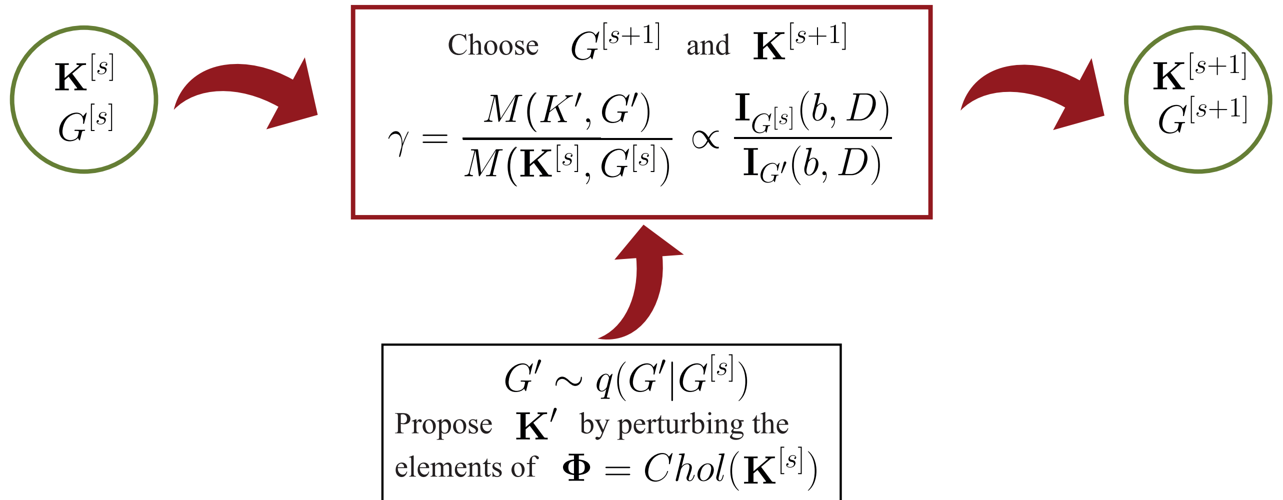

Figure 2.4. Visual representation of increasing dimension Reversible Jumps in a Gaussian graphical model framework.

2.2.4 Generalization to block graph space

In this section we want to extend Algorithm 6 to the case of multiple links changing. In particular what we propose is to map the graph G into its representative G_A. Remember that this transformation is possible only if $G \in \mathscr{B}$, which was defined in Section 1.1. Thanks to this map we can draw the proposed graph simply working in $\mathscr{G}_A$. We decide if adding or deleting one element of G_A and then we map back to $\mathscr{G}$ to obtain the proposed graph. By construction, it has for sure a block structure. It is important to underline that the addition or removal of one link in $\mathscr{G}_A$ implies, in general, a change of more than a single link in $\mathscr{G}$.
We propose the following generalization:

Step 0. *Description of the current state*
Suppose the chain to be in state $(\boldsymbol{K}^{[s]}, G^{[s]})$, with $\boldsymbol{K}^{[s]} \in \mathbb{P}_{G^{[s]}}$. Suppose moreover that $G^{[s]} \in \mathscr{B}$. This means that we are allowed to define $G_A^{[s]} = \rho^{-1}(G^{[s]}) \in \mathscr{G}_A$. We recall that for each $G_A \in \mathscr{G}_A$, we have $G_A = (V_A, E_A)$ where $V_A = \{A_1, ..., A_M\}$ is a partition of V and each A_i has cardinality $n_i \geq 1$.

Step 1. Updating the graph

1.1. *Sampling G'*
Denote with $nbd_M^+(G_A)$ the set of (multi)graphs having M nodes that can be obtained by adding an edge to $G_A \in \mathscr{G}_A$. Analogously, $nbd_M^-(G_A)$ is the set of (multi)graphs obtainable by removing an edge. Recall that if $n_i > 1$, then it is also possible to remove the diagonal element. Finally we call $nbd_M(G_A) = nbd_M^+(G_A) \cup nbd_M^-(G_A)$ the one edge way neighborhood of G_A.
Draw $G'_A \in nbd_M(G_A)$ from the proposal distribution defined in (2.5) and reported below

$$q(G'_A | G_A^{[s]}) = \frac{1}{2}\text{Unif}(nbd_M^+(G_A^{[s]})) + \frac{1}{2}\text{Unif}(nbd_M^-(G_A^{[s]}))$$

The candidate graph G' is then defined as $G' = \rho(G'_A)$. This is the same proposal we defined in Section 2.1, see Figure 2.2 for a visual representation.
Suppose, without loss of generality, that G'_A is obtained by adding edge (l_0, m_0) to $G_A^{[s]}$. We need to introduce some notations.
Denote the set of links that is changing in space $\mathscr{G}$ with

$$L = \{(i, j) : i < j, (i, j) \in E', (i, j) \notin E^{[s]}\}$$

that simply is the cartesian product of sets A_{l_0} and A_{m_0} reordered in such a way that $i < j$. Let $l = |L|$. Note that also in this framework, L cannot contain diagonal elements.
We see that $\nu(G') = \nu(G) \cup L$. We call $V(L)$ the set of vertices involved in the

change, i.e $V(L) = A_{l_0} \cup \; A_{m_0}$.
We have a simple rule for computing $d_i^{G'} - d_i^{G^{[s]}}$ and $\nu_i^{G'} - \nu_i^{G^{[s]}}$ for each $i \in V(L)$.
Suppose without loss of generality that $i \in A_{l_0}$, we then have

$$d_i^{G'} - d_i^{G^{[s]}} = |\{j \in A_{m_0} : j < i\}|$$
$$\nu_i^{G'} - \nu_i^{G^{[s]}} = |\{j \in A_{m_0} : j > i\}|$$

that is respectively the number of indices smaller than i and the number of indices grater than i in the other group.

1.2. *Construction of* $\boldsymbol{K'}$

The idea is exactly the same as for the case of only one edge changing. The difference is of course that now we have to perturb all the free elements belonging to L. We decide to do it independently and all with the same variance σ_g^2. The procedure is the following:
set $(\Phi')^{\nu(G^{[s]})} = (\Phi^{[s]})^{\nu(G^{[s]})}$, that is

$$\phi'_{ij} = \phi_{ij}^{[s]}, \quad \forall (i,j) \in E$$

Then, $\forall h \in L$ sample

$$\eta_h \sim N(\phi_h^{[s]}, \sigma_g^2) \tag{2.21}$$

and set $\phi'_h = \eta_h$.
Complete matrix Φ' through completion operation, see Theorem 2.2.2.1, and set $\boldsymbol{K}' = (\Phi')^T \Phi'$.

1.3. *Find Jacobian of RJ step*

Even if this time the change of dimension is equal to $l \geq 1$, the Jacobian is still equal to 1. Indeed the transformation is $(\ (\Phi^{[s]})^{\nu(G^{[s]})},\ \eta_1, ..., \eta_l\ ) \to (\Phi')^{\nu(G')}$. It is a map from $\mathbb{R}^{|\nu(G^{[s]})|} \times \mathbb{R}^l$ into $\mathbb{R}^{|\nu(G^{[s]})|+l}$. We still end up with a linear transformation

$$(\Phi')^{\nu(G')} = \boldsymbol{I}_{|\nu(G^{[s]})|+l}((\Phi^{[s]})^{\nu(G^{[s]})},\ \eta_1, ..., \eta_l)^T$$

1.4. *Acceptance probability*

From a theoretical point of view the terms involved are exactly the same as before. Of course we have to consider the contribution for each η_h. The fact that many links are changing has an impact also on the ratio
$J(\boldsymbol{K}' \to (\Phi')^{\nu(G')})/J(\boldsymbol{K}^{[s]} \to (\Phi^{[s]})^{\nu(G^{[s]})})$. Writing down everything explicitly we get that

$\gamma\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \rightarrow (\boldsymbol{K}', G')\right) = \min\{1, R^+\}$, where R^+ is given by

$$R^+ = \left(2\pi\sigma_g^2\right)^{l/2} \frac{\boldsymbol{I}_{G^{[s]}}(b,D)}{\boldsymbol{I}_{G'}(b,D)} \frac{\pi(G')}{\pi(G^{[s]})} \frac{|nbd_M^+(G_A^{[s]})|}{|nbd_M^-(G_A')|} \prod_{i\in V(L)} \left(\phi_{ii}^{[s]}\right)^{\nu_i^{G'} - \nu_i^{G^{[s]}}} \times$$
$$\times \exp\left\{-\frac{1}{2}\left[\langle \boldsymbol{K}' - \boldsymbol{K}^{[s]},\ D+U\rangle - \frac{1}{\sigma_g^2}\sum_{h\in L}(\eta_h - \phi_h^{[s]})^2\right]\right\} \tag{2.22}$$

In case of deletion of links, the acceptance probability is the inverse of the reverse move.
$\gamma\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \rightarrow (\boldsymbol{K}', G')\right) = \min\{1, R^-\}$ with

$$R^- = \left(2\pi\sigma_g^2\right)^{-l/2} \frac{\boldsymbol{I}_{G^{[s]}}(b,D)}{\boldsymbol{I}_{G'}(b,D)} \frac{\pi(G')}{\pi(G^{[s]})} \frac{|nbd_M^-(G_A^{[s]})|}{|nbd_M^+(G_A')|} \prod_{i\in V(L)} \left(\phi_{ii}^{[s]}\right)^{\nu_i^{G^{[s]}} - \nu_i^{G'}} \times$$
$$\times \exp\left\{-\frac{1}{2}\left[\langle \boldsymbol{K}' - \boldsymbol{K}^{[s]},\ D+U\rangle - \frac{1}{\sigma_g^2}\sum_{h\in L}(\eta_h - \phi_h^{[s]})^2\right]\right\} \tag{2.23}$$

We remaining part is exactly as before. The chain is now in $(\boldsymbol{K}^{[s+1/2]}, G^{[s+1]})$. By construction we have $\boldsymbol{K}^{[s+1/2]} \in P_{G^{[s+1]}}$.

Step 2. Updating the precision matrix
Given the updated graph $G^{[s+1]}$, draw the precision matrix

$$K^{[s+1]} \mid G^{[s+1]} \sim \text{GWishart}(b+n, D+U)$$

In this section we presented a MCMC method for navigating the space $\mathscr{B} \subset \mathscr{G}$. Thanks to the Cholesky decomposition of the precision matrix, this new chain is a natural generalization of Algorithm 6. It is so simple because the only thing that is actually changing is the proposal distribution. Note indeed that if $|A_{l_0}|= |A_{l_0}|= 1$ then also $l = 1$ which means that there is only one link changing in $\mathscr{G}$. In this case this new procedure boils down to the exact same method of Section 2.2. We call this novel procedure **Block Reversible Jumps for GGM**, Block RJ for short, and it is summarized in Algorithm 7.

Algorithm 7: Block Reversible Jumps for GGM

Suppose the chain to be in state $(\boldsymbol{K}^{[s]}, G^{[s]})$, with $\boldsymbol{K}^{[s]} \in P_{G^{[s]}}$ and $G^{[s]} \in \mathscr{B}$. For each iteration:

Step 1. Updating the graph

1.1. Sample G' from $q(G'|G^{[s]})$ given by (2.5). Suppose an addition move is selected. Deletion is analogous. Call L the set of new edges.

1.2. For each $h \in L$, draw $\eta_h \sim N(\phi_h^{[s]}, \sigma_g^2)$

1.3. Set $(\Phi')^{\nu(G^{[s]})} = (\Phi^{[s]})^{\nu(G^{[s]})}$ and $\phi'_h = \eta_h \quad \forall h \in L$.
Derive the remaining elements by completion operation and define $\boldsymbol{K}' = (\Phi')^T\Phi'$.

1.4. Compute $\gamma\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \to (\boldsymbol{K}', G')\right)$ given by (2.22).

1.5. Draw $c \sim \text{Unif}[0,1]$. **if** $c < \gamma$ **then** set $G^{[s+1]} = G'$.

Step 2. Updating the precision matrix:
Draw $\boldsymbol{K}^{[s+1]}$ from (2.20).

2.3 Eliminating evaluation of prior normalizing constant

The joint chain presented in Section 2.2 is more elaborated than the first one we derived in Section 2.1 but it defines a more scalable procedure as it does not involve any GWishart posterior normalizing constant. Nevertheless, we are not yet completely satisfied as the prior constants remain. Moreover, the acceptance ratio of our proposed block version, defined in (2.22), involves graphs that differs from more than a single link, which denies us to use the Letac et al. (2017) approximation.
The removal of that term can be done by means of an adaptation of the **Exchange algorithm**, which is presented here below in a more general framework to respect its flexibility and generality.

2.3.1 The Exchange algorithm

Monte Carlo Markov chain algorithms are used to draw samples from distributions with intractable normalizing constant. For example, let us consider a very general Bayesian model where data $\boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n\}$ are modelled as

$$p(\boldsymbol{Y}|\theta) = f(\boldsymbol{Y}|\theta)$$

where f is a general density function. The model is then completed by a prior $p(\theta)$ on the unknown parameter, whose estimation is based on the possibility of drawing samples from its

posterior distribution

$$p(\theta|\boldsymbol{Y}) = \frac{f(\boldsymbol{Y}|\theta)p(\theta)}{p(\boldsymbol{Y})} \tag{2.24}$$

which is usually known only up to a constant as $p(\boldsymbol{Y})$ can not be derived, a part from simple cases when we are able to recognize a known law. Note indeed that the normalizing constant does not depend on the parameter of interest θ. A standard Metropolis Hastings procedure, see Algorithm 8, is able to sample from (2.24), indeed in the acceptance probability ratio (2.25), $p(\boldsymbol{Y})$ cancels out. In this case, we say that the posterior (2.24) is an **intractable** distribution.
The sampling strategy is instead more challenging if we consider a Bayesian model, where the

Algorithm 8: Metropolis-Hastings Algorithm

For each iteration:

1. Propose $\theta' \sim q(\theta'|\theta)$
2. Compute
$$\gamma(\theta \to \theta') = \frac{p(\theta'|\boldsymbol{Y})q(\theta|\theta')}{p(\theta|\boldsymbol{Y})q(\theta'|\theta)} \tag{2.25}$$
3. Draw $c \sim \text{Unif}[0, 1]$
4. **if** $c < \gamma$ **then** set $\theta = \theta'$

likelihood

$$p(\boldsymbol{Y}|\theta) = f(\boldsymbol{Y}|\theta)/Z(\theta) \tag{2.26}$$

depends on a normalizing constant $Z(\theta)$ that is not available in close form or that cannot be computed explicitly. In this case we are no longer assuming that f is density function but it integral over its domain is finite and it is equal to $Z(\theta)$.
The posterior distribution we want to sample from takes is then equal to

$$p(\theta|\boldsymbol{Y}) = \frac{f(\boldsymbol{Y}|\theta)p(\theta)}{Z(\theta)p(\boldsymbol{Y})} \tag{2.27}$$

As before, $p(\boldsymbol{Y})$ is not needed but this time the normalizing constant $Z(\theta)$ cannot be ignored as it is function of the unknown parameter.

Sampling from (2.27) by means of a Metropolis Hastings sampler may result very inefficient, indeed writing explicitly the acceptance ratio we get

$$\gamma(\theta \to \theta') = \frac{p(\theta'|\boldsymbol{Y})q(\theta|\theta')}{p(\theta|\boldsymbol{Y})q(\theta'|\theta)} = \frac{f(\boldsymbol{Y}|\theta')p(\theta')q(\theta|\theta')}{f(\boldsymbol{Y}|\theta)p(\theta)q(\theta'|\theta)} \frac{Z(\theta)}{Z(\theta')} \tag{2.28}$$

we note that it depends on the ratio of the non computable constants and that there is no

way to remove it. We are not free to change f because it defines our model and, even if in principle the proposal q is completely up to us to be decided, choosing it in such a way we erase $Z(\theta)$ would be extremely difficult. A possible choice could be $q(\theta'|\theta) = Z(\theta)g(\theta')$ but it is impossible to be constructed in practice without knowing Z.
In this case we say that the posterior (2.27) is a **doubly-intractable** distribution.

The brilliant idea by Møller et al. (2006) first and improved by Murray et al. (2012) is to increase the dimension of the problem introducing an auxiliary variable $\boldsymbol{w}$ who lives in the same space of $\boldsymbol{Y}$. Then, consider the augmented joint posterior distribution

$$p(\theta, \theta', \boldsymbol{w}|\boldsymbol{Y}) = p(\theta)\frac{f(\boldsymbol{Y}|\theta)}{Z(\theta)}q(\theta'|\theta)\frac{f(\boldsymbol{w}|\theta')}{Z(\theta')} \tag{2.29}$$

that is defined in such a way that, integrating out θ' and $\boldsymbol{w}$, we end up with the original distribution $p(\theta|\boldsymbol{Y})$. With a new variable coming into play we have one more degree of freedom in the sampling strategy and we can exploit this opportunity to get rid of the intractable constants.
The Exchange algorithm samples (θ, θ', w) from the augmented distribution as explained in Algorithm 9.
We would like to underline that in step 2.2 of Algorithm 9, the ratio of proposal terms cancels

Algorithm 9: Exchange Algorithm

For each iteration, draw a sample from (2.29) by:

Step 1. Update $(\theta', \boldsymbol{w})$ with a block Gibbs-sampler:

1.1. Propose $\theta' \sim q(\theta'|\theta)$

1.2. Draw an auxiliary variable $\boldsymbol{w} \sim f(\boldsymbol{w}|\theta')/Z(\theta')$ from an exact sampler.

Step 2. Update θ:

2.1. Propose θ' as new state by exchanging the role of θ and θ' in (2.29)

2.2. Compute

$$\begin{aligned}\gamma(\theta \to \theta') = \ & \frac{p(\theta', \theta, \boldsymbol{w}|\boldsymbol{Y})}{p(\theta, \theta', \boldsymbol{w}|\boldsymbol{Y})} = \\ = \ & \frac{f(\boldsymbol{Y}|\theta')p(\theta')q(\theta|\theta')}{f(\boldsymbol{Y}|\theta)p(\theta)q(\theta'|\theta)}\frac{f(\boldsymbol{w}|\theta)}{f(\boldsymbol{w}|\theta')}\end{aligned} \tag{2.30}$$

2.3. Draw $c \sim \text{Unif}[0, 1]$

2.4. **if** $c < \gamma$ **then** set $\theta = \theta'$

out because it is a symmetric move, see Wang and Li (2012) for further details. Comparing the acceptance probability of the exchange algorithm (2.30) with a standard MH procedure

(2.25), we see that this new method simply replaces the intractable normalizing constants ratio with an estimate from a single sample of the auxiliary variable:

$$\frac{Z(\theta)}{Z(\theta')} \approx \frac{f(\boldsymbol{w}|\theta)}{f(\boldsymbol{w}|\theta')}, \text{with } w \sim f(\boldsymbol{w}|\theta')/Z(\theta')$$

the only fundamental requirement is to sample $\boldsymbol{w}$ from an exact sampler.

This procedure is a valid MCMC method, which means that it samples from the correct limiting distribution, it does not imply any approximation. The mathematical details proving this statement are not extremely difficult but long and not much illuminating on the idea behind, it can be found in (Murray et al. 2012, section 4). It is instead much more interesting the metaphor they used to explain it.

Imagine that at each step the current state θ is actually holding the data $\boldsymbol{Y}$. We generate a new parameter θ' and we would like to convince θ to give up the data to the new rival parameter θ'. How can we do it? We try to deceive it, we create another auxiliary data $\boldsymbol{w}$ and offer it an exchange. We ask θ to free $\boldsymbol{Y}$ swapping it with $\boldsymbol{w}$.
θ is willing to accept the exchange only if $\boldsymbol{w}$ is a better fit for him then $\boldsymbol{Y}$, that in mathematical terms corresponds to have $f(\boldsymbol{w}|\theta)/f(\boldsymbol{Y}|\theta) > 1$.
Of course we need to consider both sides, which means that the swap has to be fair also for θ'. Since it is getting $\boldsymbol{Y}$ in place of $\boldsymbol{w}$, the ratio $f(\boldsymbol{Y}|\theta')/f(\boldsymbol{w}|\theta')$ takes into account how much θ' prefers $\boldsymbol{Y}$ with respect to $\boldsymbol{w}$.
The remaining terms in (2.30) reflect any disparity between the frequency with which we propose swaps and our prior beliefs over the parameter that should own the data.
Of course this is only the idea behind and not a proof but it gives a clear interpretation of what is happening and why it is called Exchange algorithm. The key point is that if we have an exact sampler for $f(\boldsymbol{w}|\theta)/Z(\theta)$ we are then able to avoid the computation of the ratio $Z(\theta')/Z(\theta)$ and it is possible to do it in an way that generates a valid MCMC.

2.3.2 Double Reversible Jumps MCMC for GGM

We now want to mimic the arguments behind the Exchange algorithm in order to get rid of the ratio $\boldsymbol{I}_{G^{[s]}}(b, D)\ /\ \boldsymbol{I}_{G'}(b, D)$. In particular we would also need to extend it to a trans-dimentional case. Algorithm 9 is thought to generalize a standard Metropolis Hastings procedure where θ and θ' belong to the same space while our need is to extend a Reversible Jump chain.

In a graphical model framework, G is playing a role similar to the one of θ and $\boldsymbol{K}$ the one of $\boldsymbol{Y}$.

$$P(\boldsymbol{K}|G) = \frac{f(\boldsymbol{K}|G)}{I_G(b, D)} \tag{2.31}$$

where

$$f(\boldsymbol{K}|G) = |\boldsymbol{K}|^{\frac{b-2}{2}} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K}D\right)\right\} \tag{2.32}$$

and

$$I_G(b, D) = \int_{\mathbb{P}_G} |\boldsymbol{K}|^{\frac{b-2}{2}} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K}D\right)\right\} \, d\boldsymbol{K} \tag{2.33}$$

is the doubly-intractable constant we called $Z(\theta)$ in Section 2.3.1.
Comparing (2.31) and (2.27) we see that in this case the undesired constant is not part of the likelihood but of prior. This does not cause any special difficulty, the only difference is the presence of an extra term, that is the ratio of the likelihoods which does not play any important role in this extension.

Suppose as usual that the chain is in the state $(\boldsymbol{K}^{[s]}, \boldsymbol{G}^{[s]})$ and we are proposing $\boldsymbol{G}'$ by adding edge (i_0, j_0) to the current graph.
Applying the same idea of the previous section, we can think of $\boldsymbol{K}^{[s]}$ as if it is currently held by $\boldsymbol{G}^{[s]}$. We would like to steal the precision matrix from the graph and give it the new one $\boldsymbol{G}'$. To do that, we try to deceive $\boldsymbol{G}^{[s]}$ by proposing a swap. We offer it $\boldsymbol{W}^0 \in \mathbb{P}_{G^{[s]}}$ in exchange of $\boldsymbol{K}^{[s]}$. This particular swap has to be convenient for the current graph, which happens if

$$\frac{P(\boldsymbol{W}^0|\boldsymbol{G}^{[s]})}{P(\boldsymbol{K}^{[s]}|\boldsymbol{G}^{[s]})} = \frac{I_{G^{[s]}}(b, D)}{I_{G^{[s]}}(b, D)} \frac{f(\boldsymbol{W}^0|\boldsymbol{G}^{[s]})}{f(\boldsymbol{K}^{[s]}|\boldsymbol{G}^{[s]})} > 1 \tag{2.34}$$

Note that in (2.34), $I_{G^{[s]}}(b, D)$ cancels out.
This is only one half of the exchange process, we now have to see it from the point of view of $\boldsymbol{G}'$. This has to be done with a little bit of care because $\boldsymbol{W}^0$ and $\boldsymbol{K}^{[s]}$ do not belong to $\mathbb{P}_{G'}$, this is where we need to slightly modify Algorithm 9 and adapt it to dimension changing moves. We first need to construct the proposed matrices in the right spaces, respectively $\tilde{\boldsymbol{K}}, \boldsymbol{K}' \in \mathbb{P}_{G'}$ and then we see if $\boldsymbol{G}'$ is willing to accept the exchange by evaluating

$$\frac{P(\boldsymbol{K}'|G')}{P(\tilde{\boldsymbol{W}}|G')} = \frac{I_{G'}(b, D)}{I_{G'}(b, D)} \frac{f(\boldsymbol{K}'|G')}{f(\tilde{\boldsymbol{W}}|G')} \tag{2.35}$$

This ratio erases the second intractable constant.
The crucial point in the algorithm is the construction of those four matrices. Here is the

	$G^{[s]}$	G'
Actual precision matrix	$\boldsymbol{K}^{[s]}/\Phi^{[s]}$	$\boldsymbol{K}'/\Phi'$
Auxiliary precision matrix	$\boldsymbol{W}^0/\Phi^0$	$\tilde{\boldsymbol{W}}/\tilde{\Phi}$

Table 2.1. The four involved precision matrices and their Cholesky decomposition.

procedure proposed by Lenkoski (2013).

1. *Construction of* $\tilde{\boldsymbol{W}}$
   Sample $\tilde{\boldsymbol{W}}|G' \sim \text{GWishart}(b, D)$ from an exact sampler. Note that having an exact sampler is fundamental at this stage, otherwise the following construction is not valid.

2. *Construction of* $\boldsymbol{K}'$
   Sample $\eta \sim N(\phi^{[s]}_{i_0 j_0}, \sigma^2_g)$. Set $(\Phi')^{\nu(G^{[s]})} = (\Phi^{[s]})^{\nu(G^{[s]})}$ and $\phi'_{i_0 j_0} = \eta$. Derive the remaining elements by completion operation and define $\boldsymbol{K}' = (\Phi')^T \Phi'$.

3. *Construction of* $\boldsymbol{W}^0$
   Set $(\Phi^0)^{\nu(G^{[s]})} = (\tilde{\Phi})^{\nu(G^{[s]})}$. This operation assigns a value to each free element of Φ^0, so all the others are determined by completion. In particular note that $\tilde{\phi}_{i_0 j_0}$ is a free element while $\phi^0_{i_0 j_0}$ is not.

The joint augmented distribution we sample from is

$$P(\boldsymbol{K}, G, \tilde{\boldsymbol{W}}, G' \mid \boldsymbol{Y}) = P(\boldsymbol{K}, G \mid \boldsymbol{Y}) q(G'|G^{[s]}) P(\tilde{\boldsymbol{W}}, G' \mid \boldsymbol{Y}) \tag{2.36}$$

that is the analogous of (2.29), the first term is the target distribution, i.e the posterior of model (1.9), then the proposal and at the end there is the augmented term.
The way we sample from (2.36) is called **Double Reversible Jumps for GGM**, DRJ for short, because it considers switching between

$$(\boldsymbol{K}, G, \tilde{\boldsymbol{W}}, G')$$

to the alternative

$$(\boldsymbol{K}', G', \boldsymbol{W}^0, G)$$

by performing two Reversible Jump moves; a dimension increasing step from $(\boldsymbol{K}, G)$ to $(\boldsymbol{K}', G')$ according to the posterior parameters $b + n$ and $D + U$ and a dimension decreasing jump from $(\tilde{\boldsymbol{W}}, G')$ to $(\boldsymbol{W}^0, G)$ according to the prior parameters b and D. If we apply twice the procedure explained in Algorithm 6 we find out that the probability of accepting the move is equal to
$\gamma'\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \to (\boldsymbol{K}', G')\right) = \min\{1, R^+\}$, where

$$\begin{aligned} R^+ = {} & \frac{\pi(G')}{\pi(G^{[s]})} \frac{\phi^{[s]}_{i_0 i_0}}{\phi^0_{i_0 i_0}} \frac{|nbd^+_p(G^{[s]})|}{|nbd^-_p(G')|} \frac{\exp\left\{-\frac{1}{2}\langle \boldsymbol{K}' - \boldsymbol{K}^{[s]},\ D + U\rangle\right\}}{\exp\left\{-\frac{1}{2}\langle \tilde{\boldsymbol{W}} - \boldsymbol{W}^0,\ D\rangle\right\}} \times \\ & \times \exp\left\{\frac{(\phi'_{i_0 j_0} - \phi^{[s]}_{i_0 j_0})^2 - (\phi^0_{i_0 j_0} - \tilde{\phi}_{i_0 j_0})^2}{2\sigma^2_g}\right\} \end{aligned} \tag{2.37}$$

Once the next graph has been decided, the precision matrix is sampled as before from (2.20).

It is clear that (2.37) is much more appealing than (2.18). It shows that the only price we have to pay for avoiding computing the ratio $\boldsymbol{I}_{G^{[s]}}(b, D)\ /\ \boldsymbol{I}_{G'}(b, D)$ is drawing an auxiliary matrix and then performing some algebraic operations. The gain is not only in computational effort but also that the result is theoretically correct and not an approximation, as if we used a Monte Carlo method to evaluate the GWishart normalizing constant. The resulting method is summarized in Algorithm 10.

Algorithm 10: Double Reversible Jumps for GGM

Suppose the chain to be in state $(\boldsymbol{K}^{[s]}, G^{[s]})$, with $\boldsymbol{K}^{[s]} \in P_{G^{[s]}}$ and $G^{[s]} \in \mathscr{B}$. For each iteration:

Step 1. Updating the graph

1.1. Sample G' from $q(G'|G^{[s]})$ given by (2.2). Suppose link (i_0, j_0) has to be added. Deletion is analogous.

1.2. Draw $\tilde{\boldsymbol{W}}|G' \sim$ GWishart(b, D) from an exact sampler.

1.3. Draw $\eta \sim N(\phi^{[s]}_{i_0 j_0}, \sigma^2_g)$.

1.4. Set $(\Phi')^{\nu(G^{[s]})} = (\Phi^{[s]})^{\nu(G^{[s]})}$ and $\phi'_{i_0 j_0} = \eta$.
Derive the remaining elements by completion operation and define $\boldsymbol{K}' = (\Phi')^T \Phi'$.

1.5. Set $(\Phi^0)^{\nu(G^{[s]})} = (\tilde{\Phi})^{\nu(G^{[s]})}$. Derive the remaining elements by completion operation and define $\boldsymbol{W}^0 = (\Phi^0)^T \Phi^0$.

1.6. Compute $\gamma\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \to (\boldsymbol{K}', G')\right)$ given by (2.37).

1.7. Draw $c \sim$ Unif$[0, 1]$. **if** $c < \gamma$ **then** set $G^{[s+1]} = G'$.

Step 2. Updating the precision matrix:
Draw $\boldsymbol{K}^{[s+1]}$ from (2.20).

2.3.3 Block Double Reversible Jumps MCMC for GGM

We are finally ready to present our proposed Monte Carlo Markov chain over the space $\mathscr{B}$ of block graphs. It is obtained by enriching Algorithm 7 with a trans-dimensional extension of the Exchange algorithm, the same way we derived Algorithm 10 starting from Algorithm 6. We called the resulting method **Block Double Reversible Jumps for GGM**, or Block DRJ for short. It is summarized in Algorithm 11. The same notation of the previous sections is applied.

Algorithm 11: Block Double Reversible Jumps for GGM

Suppose the chain to be in state $(\boldsymbol{K}^{[s]}, G^{[s]})$, with $\boldsymbol{K}^{[s]} \in P_{G^{[s]}}$ and $G^{[s]} \in \mathscr{B}$. For each iteration:

Step 1. Updating the graph

1.1. Sample G' from $q(G'|G^{[s]})$ given by (2.5). Suppose an addition move is selected. Deletion is analogous. Call L the set of new edges.

1.2. Draw $\tilde{\boldsymbol{W}}|G' \sim \text{GWishart}(b, D)$ from an exact sampler.

1.3. For each $h \in L$, draw $\eta_h \sim N(\phi_h^{[s]}, \sigma_g^2)$

1.4. Set $(\Phi')^{\nu(G^{[s]})} = (\Phi^{[s]})^{\nu(G^{[s]})}$ and $\phi'_h = \eta_h \ \ \forall h \in L$.
Derive the remaining elements by completion operation and define $\boldsymbol{K}' = (\Phi')^T\Phi'$.

1.5. Set $(\Phi^0)^{\nu(G^{[s]})} = (\tilde{\Phi})^{\nu(G^{[s]})}$. Derive the remaining elements by completion operation and define $\boldsymbol{W}^0 = (\Phi^0)^T\Phi^0$.

1.6. Compute $\gamma\left(\left(\boldsymbol{K}^{[s]}, G^{[s]}\right) \to (\boldsymbol{K}', G')\right) = \min\{1, R^+\}$ where

$$
\begin{aligned}
R^+ = & \frac{\exp\left\{-\frac{1}{2}\langle \boldsymbol{K}' - \boldsymbol{K}^{[s]},\ D + U\rangle\right\}}{\exp\left\{-\frac{1}{2}\langle \tilde{\boldsymbol{W}} - \boldsymbol{W}^0,\ D\rangle\right\}} \frac{\pi(G')}{\pi(G^{[s]})} \prod_{i \in V(L)} \left(\frac{\phi_{ii}^{[s]}}{\phi_{ii}^0}\right)^{\nu_i^{G'} - \nu_i^{G^{[s]}}} \\
& \exp\left\{\frac{1}{2\sigma_g^2} \sum_{h \in L} \left[\left(\phi'_h - \phi_h^{[s]}\right)^2 - \left(\phi_h^0 - \tilde{\phi}_h\right)^2\right]\right\}
\end{aligned}
\tag{2.38}
$$

1.7. Draw $c \sim \text{Unif}[0,1]$. **if** $c < \gamma$ **then** set $G^{[s+1]} = G'$.

Step 2. Updating the precision matrix:
Draw $\boldsymbol{K}^{[s+1]}$ from (2.20).

Wrapping up, in this section we presented six different Monte Carlo Markov chain methods for Gaussian graphical models. Three of them, ARMH Algorithm 4, RJ for GGM Algorithm 6 and DRJ for GGM Algorithm 10 are taken from the literature and they explore the whole space of possible graphs, $\mathscr{G}$. The need of presenting three different procedure is due to the presence of the intractable GWishart normalizing constant. The ARMH is the simplest and most intuitive algorithm but, because of the presence of posterior normalizing constant, it is not a scalable solution.
To show its limitations, we performed the following test: we simulated data form model (1.9) using different values for p that is the dimension of the graph. In particular we considered $p = 6, 8, 10, 15$ and, for each possibility, we simulated 25 datasets so that we can be reasonably sure that the outputs are not biased because of the particular choice of the graph. Given the dataset, we run three different chains, one for every MCMC method previously presented. All computational details are given in Chapter 3. The number of iterations performed depends on

the dimension of the graph. In the first case, 1000 iterations plus 1000 more as burnin period are enough to obtain sharp estimates, in the other cases we made 5000, 50000 and 100000 plus the same number as burinin. For this test, we refer to the Structural Hamming Distance (SHD, Tsamardinos et al. 2006) to evaluate the ability of recovering the underlying graph. It simply represents the number of edge insertions and deletions needed to transform the estimated graph into the true graph, therefore lower values of SHD correspond to better performances. To compare different setting, the SHD has been standardized over the maximum number of edges, that we recall to be equal to $\binom{p}{2}$. The resulting values are summarized in Figure 2.5.

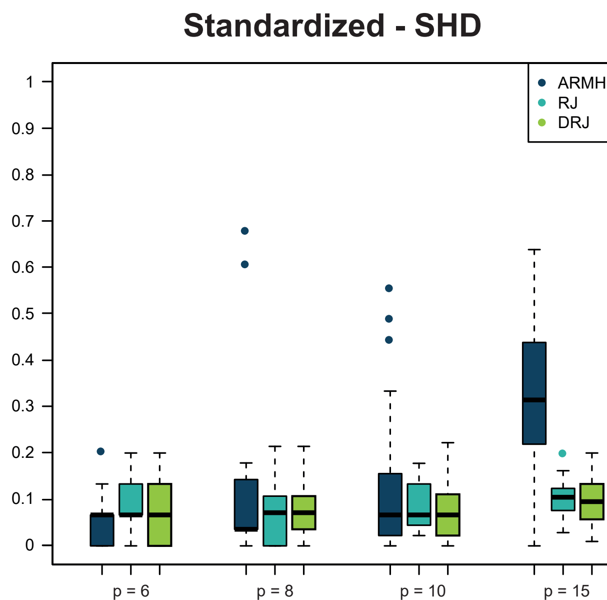

Figure 2.5. Experiment assessing the non scalability of the ARMH algorithm. For different values of p, we simulated 25 datasets and run the analysis using the three different MCMC methods (ARMH, RJ and DRJ). For each simulation we computed the Structural Hamming Distance between the estimated graph and the true one, standardized with respect to the number of links to be estimated.

For $p = 6$ the most precise method is the ARMH. The number of iterations performed in this case is very limited, only 2000 taking into account also the burnin period. The complexity of the Reversible Jumps methods (RJ and DRJ) slows down their convergence while the ARMH only need to compare the marginal posterior distribution of not many graphs, hence it converges rapidly. Moreover in Section 1.3.1 we already showed that for $p = 6$ the approximation for the normalizing constants is very accurate.

The cases $p = 8$ and $p = 10$ are kind of intermediate situations: in the first one the median line of the ARMH boxplot is slightly lower than the other two and for $p = 10$ they are almost at the same level. Nevertheless the variability is higher, suggesting that it may present a more unstable behaviour with respect to the competitive methods. Those suspicions are confirmed when we observe the $p = 15$ case which clarifies that the ARMH is not reliable as soon as the dimension of the problem is not trivial. The Reversible Jumps approach is more scalable also

from the computational point of view. For the ARMH, the $p = 15$ case took almost 25 hours to accomplish all the 25 simulations while the others were able to finish in less than 3 hours. This is also a significant example to see the impact of the combinatorial speed of growth for the size of $\mathscr{G}$, just look at how different the number of misclassified links can be when adding just 5 nodes to the graph.

Our work consists in generalizing those methods to investigate only a smaller subset of that space, that is the one composed of block structured graphs only, $\mathscr{B}$. The resulting methods, called Block ARMH Algorithm 5, Block RJ for GGM Algorithm 7 and Block DRJ for GGM Algorithm 11 perform the search by adding or removing a full block at each iteration. The last one, Block Double Reversible Jumps for GGM, avoids any computation of the GWishart normalizing constant which makes it suitable for dealing with high dimensional problems.

2.4 Comparison with similar frequentist approach

We already mentioned that we are not aware of any similar Bayesian approach, at least for that part of the literature that makes use of a GWishart prior distribution for the precision matrix. Instead, looking at the frequentist statistics paradigma, we found out some methods that aim to impose a structure on the graph. First of all, we follow Tan et al. (2015) to frame the structural leaning in the frequentist scenario.

Let $\tilde{\boldsymbol{Y}}$ be the $n \times p$ matrix collecting the n observation of p dimensional features. Its rows are denoted as $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$ and assumed to be $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n \overset{\text{iid}}{\sim} N_p(\boldsymbol{0}, \boldsymbol{K})$. As usual $\boldsymbol{K}$ is a precision matrix and we here call $\boldsymbol{\Sigma} = \boldsymbol{K}^{-1}$ the covariance matrix. Moreover, let $\boldsymbol{S} \in \mathbb{R}^{p \times p}$ be the empirical covariance matrix, defined as $\boldsymbol{S} = \tilde{\boldsymbol{Y}}^T \tilde{\boldsymbol{Y}} / n$ and $\mathbb{S}^p_{++}$ the space of all positive definite matrices of size p. A remarkable difference is that in our Bayesian framework we always identified the graph G with its adjacency matrix. It is not the only algebraic representation, we stress this fact by calling G the underlying graph and denoting with $\boldsymbol{\Theta} \in \mathbb{R}^{p \times p}$ the graph matrix, whose elements are not necessarily binary values.
A natural way to estimate $\boldsymbol{K}$ is via maximum likelihood, which involves maximizing

$$\begin{aligned} \underset{\boldsymbol{\Theta}}{\text{maximize}} \quad & \text{logdet}\,(\boldsymbol{\Sigma}) - tr\,(\boldsymbol{S}\boldsymbol{\Theta}) \\ \text{subject to} \quad & \boldsymbol{\Theta} \in \mathbb{S}^p_{++} \end{aligned} \tag{2.39}$$

The solution $\hat{\boldsymbol{\Theta}} = \boldsymbol{S}^{-1}$ is a possible estimate for $\boldsymbol{K}$ but in general it is not a good one. $\boldsymbol{S}^{-1}$ typically contains no elements that are exactly equal to zero. This corresponds to a graph in which the nodes are fully connected to each other, it does not provide useful information. Yuan and Lin (2007) proposed to introduce a ℓ_1-norm regularization on the off diagonal elements of $\boldsymbol{\Theta}$, which promotes element-wise sparsity on the graph matrix $\boldsymbol{\Theta}$. (2.39) is the reformulated as

$$\begin{aligned} \underset{\Theta}{\text{maximize}} \quad & \text{logdet}\,(\boldsymbol{\Sigma}) - tr\,(\boldsymbol{S\Theta}) - \lambda \sum_{j \notin j'} |\boldsymbol{\Theta}_{ij}| \\ \text{subject to} \quad & \boldsymbol{\Theta} \in \mathbb{S}^{p}_{++} \end{aligned} \tag{2.40}$$

Friedman et al. (2008) proposed an efficient algorithm, known as **GLasso**, to solve (2.40), hence we will refer to its maximizer a *grahical lasso solution*, or GLasso solution for short. It serves as an estimates for $\boldsymbol{K}$ and, when the nonnegative tuning parameter λ is sufficiently large, the estimated matrix will be sparse, with some elements exactly equal to zero.
The GLasso solution can be seen as the analogue estimate for the precision matrix for the Bayesian model (1.9), at least when the graphical search if performed on the space of all complete graphs. Our proposed Algorithm 11 aims instead to impose a particular structure on the graph. To achieve a similar outcome, solving (2.40) is not enough as it only enforces a uniform sparsity (Kumar et al. 2020). In other words, we do not compare ourselves with the GLasso algorithm but it is the building block for two other methods that are reasonably similar to ours.

The first one we present is the **Cluster Graphical Lasso** (CGL) proposed by Tan et al. (2015) and reported in Algorithm 12. $\tilde{\boldsymbol{S}}$ is defined such that $\tilde{\boldsymbol{S}}_{ij} = |\boldsymbol{S}_{ij}|$.

Algorithm 12: Cluster Graphical Lasso (CGL)

Step 1. Let $C_1, \dots, C_K$ be the clusters obtained by performing a clustering method of the choice based on the similarity matrix $\tilde{\boldsymbol{S}}$.

Step 2. For each $k = 1, \dots, K$:

- **2.1** Let $\boldsymbol{S}_k$ be the empirical covariance matrix for the features in the k-th cluster. It is a $|C_k| \times |C_k|$ matrix.
- **2.2** Solve the GLasso problem (2.40), using the covariance matrix $\boldsymbol{S}_k$. Let $\hat{\boldsymbol{\Theta}}_k$ be the GLasso solution.

Step 3. Combine the K resulting graphical lasso estimates into a $p \times p$ matrix $\hat{\boldsymbol{\Theta}}$, that is block diagonal with block given by $\hat{\boldsymbol{\Theta}}_1, \dots, \hat{\boldsymbol{\Theta}}_K$.

Our solution and the one obtained by means of the CGL procedure may share similar structures but only for those elements close the the diagonal, as both are able to induce a block diagonal structure. Within those blocks, the CGL is more precise since it induces sparsity within each block while we force them to be complete or empty. From another hand, Algorithm 12 does not allow for long term interactions, i.e relationships among different clusters, which can be very interesting and that are instead naturally considered in our work. They are indeed links between nodes of our the multigraph representation presented in Chapter 1. The

structure imposed by the CGL is called k-components and it is defined later in this work. See Figure 2.6 for a visual representation. The number k of disconnected components is not to be fixed but it is estimated in the first step of Algorithm 12.

The second approach we compare to is to one presented in Kumar et al. (2020) where a new formulation of the problem is given, so that the structured graph learning performed by imposing analytical spectral constraints on Laplacian and adjacency matrices. The concept of Laplacian matrix has not yet been introduced, we then need to take a step back to clarify the problem formulation.

They consider an undirected graph $G = (V, E)$ having no self-loops or multiple edges but its links are associated to positive weights $w_{ij} > 0$. Our algebraic representation of the graph has always been based on the adjacency matrix, they instead stress the fact that another representation is indeed possible, the so called Laplacian matrix.

A matrix $\mathbf{\Theta} \in \mathbb{R}^{p\times p}$ is called Laplacian matrix if its elements satisfy

$$\mathbb{S}_{\mathbf{\Theta}} = \left\{ \mathbf{\Theta} \mid \mathbf{\Theta}_{ij} = \mathbf{\Theta}_{ji} \leq 0 \text{ for } i \neq j; \; \mathbf{\Theta}_{ii} = \sum_{j\neq i} \mathbf{\Theta}_{ij} \right\} \tag{2.41}$$

Definition (2.41) implies that $\mathbf{\Theta}$ is diagonally dominant, i.e $|\mathbf{\Theta}_{ii}|= |\sum_{j\neq i} \mathbf{\Theta}_{ij}|$ and therefore positive definite (den Hertog et al. 1993, Proposition 2.2.20) having non-positive off-diagonal elements. Finally note that its rows (and columns, being symmetric) sum to zero, $\mathbf{\Theta}\mathbf{1} = \mathbf{0}$. It is related to the weighted adjacency matrix, here denoted by $\mathbf{\Theta}^A$, by the following relationship:

$$\mathbf{\Theta}^A_{ij} = \begin{cases} -\mathbf{\Theta}_{ij}, & \text{if } i \neq j \\ 0, & \text{if } i = j \end{cases} \tag{2.42}$$

implying that also for $\mathbf{\Theta}$, a null element means no connectivity between vertices i and j.

The authors in Kumar et al. (2020) present a general framework to learn the graph as a Laplacian matrix $\mathbf{\Theta}$ under some eigenvalue constraints motivated by prior information. The general optimization problem to be solved is

$$\begin{aligned} \underset{\mathbf{\Theta}}{\text{maximize}} \quad & \log \operatorname{gdet}(\mathbf{\Theta}) - tr\,(\mathbf{\Theta}\boldsymbol{S}) - \lambda h\,(\mathbf{\Theta}) \\ \text{subject to} \quad & \mathbf{\Theta} \in \mathbb{S}_{\mathbf{\Theta}}, \; \mathbf{\Lambda}_{\mathrm{T}(\mathbf{\Theta})} \in \mathbb{S}_{\mathrm{T}} \end{aligned} \tag{2.43}$$

where $\boldsymbol{S}$ is the empirical covariance matrix, $\operatorname{gdet}(\cdot)$ is the generalized determinant, defined as the product of non-zero eigenvalues, $h(\cdot)$ is the regularization term, for example one can choose the ℓ_1-norm to obtain a penalization as in equation (2.40) and λ its tuning parameter. For what concerns the constraints sets, $\mathbb{S}_{\mathbf{\Theta}}$ has already been introduced in (2.41), $\mathbf{\Lambda}_{\mathrm{T}(\mathbf{\Theta})}$ denotes the set of the eigenvalues of $\mathrm{T}\,(\mathbf{\Theta})$, that is a transformation of $\mathbf{\Theta}$: if T is the identity ($\mathrm{T}\,(\mathbf{\Theta}) = \mathbf{\Theta}$), the spectral constraints are imposed on the Laplacian matrix $\mathbf{\Theta}$, while the second possibility

is to choose $\mathrm{T}(\mathbf{\Theta}) = \mathbf{\Theta}^A$ and hence enforcing constraints on the eigenvalues of the adjacency matrix. Those constraints are are defined in the set $\mathbb{S}_{\mathrm{T}}$. The idea of (2.43) is to learn the structure of the graph from the given data statistic $\boldsymbol{S}$ by enforcing the solution $\mathbf{\Theta}$ to be a Laplacian matrix since it has to belong to $\mathbb{S}_{\mathbf{\Theta}}$ and imposing spectral constraints via T. An example will clarify how.

Consider a *k-component* connected graph, which means that its vertex set V can be partitioned in k disjoint subsets $\{V_i\}_{i=1}^k$ such that no edge connect two different components. See Figure 2.6 for a simple example having $k = 3$. This k-component structural property is directly encoded in the eigenvalues of its Laplacian matrix, indeed the number of connected components k is equal to the multiplicity of the zero eigenvalue of $\mathbf{\Theta}$ (Chung and Graham 1997). This spectral constrain can be exploited to impose this particular structure to the graph by solving the following optimization problem:

$$\begin{aligned} \underset{\mathbf{\Theta},\boldsymbol{\lambda},U}{\text{maximize}} \quad & \log \operatorname{gdet}(\mathbf{\Theta}) - tr\left(\mathbf{\Theta}\boldsymbol{S}\right) - \lambda h\left(\mathbf{\Theta}\right) \\ \text{subject to} \quad & \mathbf{\Theta} \in \mathbb{S}_{\mathbf{\Theta}},\ \mathbf{\Theta} = U \operatorname{diag}(\boldsymbol{\lambda}) U^T,\ U^T U = \boldsymbol{I}_p,\ \boldsymbol{\lambda} \in \mathbb{S}_\lambda \end{aligned} \tag{2.44}$$

where $\boldsymbol{\lambda} = \{\lambda_i\}_{i=1}^p$ are the eigenvalues of $\mathbf{\Theta}$ and $U \in \mathbb{R}^{p\times p}$ its eigenvectors matrix. The constraints $\mathbf{\Theta}$ enforces the matrix to be Laplacian and the k-component structure can be achieved by choosing $\mathbb{S}_\lambda$ so that the null eigenvalue has multiplicity equal to k.

$$\mathbb{S}_\lambda = \left\{ (\lambda_j = 0)_{j=1}^k,\ c_1 \le \lambda_{k+1} \le \cdots \le \lambda_p \le c_2 \right\} \tag{2.45}$$

The k-component structure is not the only one that can be achieved by spectral constrains. Other possibilities include *regular graphs*, *multi-component regular graphs*, *bipartite graphs*, *multi-component bipartite-regular graphs*. Their proper definitions can be found in Kumar et al. (2020), our concern here is that they can all be represented by imposing constraints on the eigenvalues of $\mathbf{\Theta}$ or $\mathbf{\Theta}^A$ or both, similarly to what we showed in (2.45). Note that the final estimate of CLG algorithm is a k-component graph, the difference is that the number of clusters is unknown while in (2.45) has to be fixed. The authors pointed out that even if it is wrongly chosen, for example $k_{true} > k_{fix}$, their algorithm is forced to identify only k_{fix} disconnected components but the edges among some of them are actually so few that one is able to recognize that the real number of clusters is indeed equal to k_{true}.

The way we tackle the problem of searching structured graphs is significantly different from (2.43) and not only because of the contrast between Bayesian and frequentist statistics. First of all, the graphical setting is not the same: we keep separated the role of the precision matrix and the undirected, unweighted graph that describes its structure while in Kumar et al. (2020) a weighted graph is exploited. Those weights represent how similar the two nodes are but they are forced to be positive, meaning that positive partial correlation among all pairs of variables

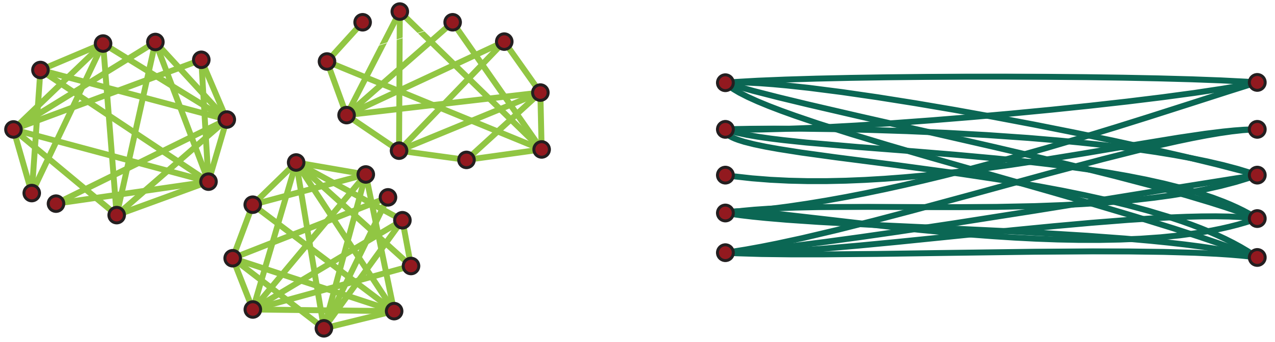

Figure 2.6. A k-component graph (left) with $k = 3$, 10 nodes in each component and a bipartite graph (right) composed of two groups having 5 vertices each.

is assumed a priori (Liu et al. 2014), which may be unrealistic in real world applications. This is a common restriction of those methods based on the Laplacian matrix of the graph (2.41). Despite technical differences, the final aim is very similar as they both try to generalize existing methods not to impose only uniform sparsity in the graph but a more elaborated structure. In (2.43) it is translated in analytical spectral constraints, which implies that the desired structure (1) has to chosen a priori, (2) a theoretical result translating it into spectral constraints has to be available and (3) the solution is forced to be of that particular form. A deep knowledge of the problem is required to decide in advance what type of solution is desired but, when it is available, this method is able to guarantee it.
The type of structure we look for can not be represented by the previously cited ones. Our goal is to divide the variable into groups and to study dependence between and within groups. The within structure could be represented by means of a multi-component graph but in that case no interaction between groups is allowed. To represent that, the closest one is probably the bipartite graphs, but it would only allow to have two groups such that the nodes inside each groups are completely disconnected. A simple example is given in Figure 2.6. Moreover, we do not fix any structure a priori, gaining more flexibility with respect to (2.43) but on the other hand it enforces a uniformly sparse structure among groups, it is unlikely to obtain exactly a multi-component or a bipartite graph. Once again, we recall that we only account for complete interactions, if link two groups, we force all nodes of the first one to be connected with all the ones of the second group. This hypothesis has been done to derived a simpler problem, i.e to reduce the space where the solution has to be searched. We justified this assumption by noting that in big data applications an overview of the underlying structure is more meaningful than all considering all pairs relationships that would be difficult to be interpreted. If this is a reasonable hypothesis depends on the specific situation.

To conclude, we see the two approaches as complementary rather then conflicting. The problem of obtaining structured solutions has been scratched only recently and it is far from being solved. If many prior information about the structure are available (2.43) is the best

choice. If one does not want to impose such rigid conditions or when the main focus is about how different groups collaborate one another, our Block search Algorithm 11 is more suitable.

2.5 Application to functional data analysis

As we have anticipated in the Introduction, Gaussian graphical models can be applied to a huge variety of problems. The application field we are interested in is *functional data analysis*, where we would like to combine a traditional smoothing procedure with Gaussian graphical modeling for studying possible dependencies among different portion of the domain of interest. In particular we are motivated by the analysis of spectrographic data, whose precise description is postponed to Section 4.5 as we here prefer to remain as general as possible.
We want to stress that our final goal is not to provide an estimate for the correlation or covariance structure of the functional data, i.e the relationships between curve, but we look for a common conditional dependence pattern within curves, which translates in frequency bands correspondence as we embed our model in a spectrographic analysis scenario. Even though the first problem has been deeply studied in the literature (Qiao et al. 2019; Zhu et al. 2016) we are not aware of any similar work that focuses on making inference on such structure.
Our contribution is therefore twofold, from the point of view of graphical models, the novelty is that the graph is no longer placed on data but on unknown parameters while for functional data analysis we introduce the study of interactions among different portions of the curve. To properly justify the previous sentence we first need to revise some basic concept of functional data analysis (Ramsay and Silverman 2005, Chapter 3).
In real life, given n observations, the i-th functional datum can only be measured on a discrete

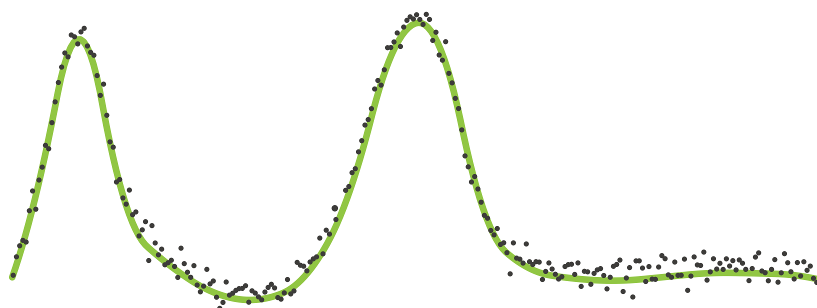

Figure 2.7. Noisy, equally spaced, measurements (black dots) and their functional, smoothed, representation (green solid line).

grid of r points, $\boldsymbol{y}_i = \{y_{i,1}, \ldots, y_{i,r}\}$. The first task is to recover its continuous representation by converting these values to function $y_i(s)$ computable for any desired argument value s. The first choice to be made is the functional space we want to embed data into, we here focus on $\mathrm{L}^2[l, u]$ as we are not concerned about derivatives. Since it is a separable Hilbert space, every function in that space can be represented as an infinite linear combination of basis functions.

For reasons that will soon be clear, we here consider a B-spline basis, denoted by $\{\varphi_j\}_{j\in\mathbb{N}}$. Of course in practice we can not construct an infinite amount of basis functions but we know that every L^2-element can be approximated arbitrarily well by taking a linear combination of a sufficiently large number p of basis elements.

$$y_i(s) = \sum_{j=1}^{p} \beta_{ij}\varphi_j(s), \quad \forall s \in [l, u]$$

If the discrete values are assumed to be errorless, we can take $p = r$ to obtain an exact representation, or *interpolation*, in the sense that we can take the coefficients β_{ij} so that the functional representation passes through the measured values, $y_i(s_k) = y_{i,k}$ for each $k = 1, \ldots, r$. When measurements are affected by noise that has to be removed, we chose $p < r$ leading to the so called **smoothing** technique, here framed within an appropriate Bayesian model.

Let n be the number of available curves, r the number of grid points $\{s_1, \ldots, s_r\}$, which is the size of the common discrete grid where those curves were evaluated, and p be the number of spline bases chosen for the smoothing.
For each $i = 1, \ldots, n$, let $\boldsymbol{y}_i = \{y_{i,1}, \ldots, y_{i,r}\}$ the measured values for the i-th curve and let $y_i(s)$ be the value at point $s \in [l, u]$ for its, unknown, functional representation. We assume that $[l, u] \subset \mathbb{R}$ is the common domain of all curves and the measurements are taken on the same grid. Finally, we denote by $\boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n\}$ the set of all observed data. According to the usual smoothing technique, the model we assume for the i-th functional data is:

$$\boldsymbol{y}_i = \boldsymbol{\Phi}\beta_i + \boldsymbol{\varepsilon}_i \tag{2.46}$$

where $\varphi_1, \ldots, \varphi_p$ are suitable cubic B-spline basis, $\boldsymbol{\beta}_i = (\beta_{i1}, \ldots, \beta_{ip})^\top$ is the spline coefficients' vector specific to i-th curve and $\boldsymbol{\varepsilon}_i \in \mathbb{R}^r$ is the measurement error vector. In (2.46) we also introduced the B-spline design matrix $\boldsymbol{\Phi} \in \mathbb{R}^{r\times p}$, that is defined as follow:

$$\boldsymbol{\Phi} = \begin{bmatrix} \varphi_1(s_1) & \varphi_2(s_1) & \ldots & \varphi_p(s_1) \\ \varphi_1(s_2) & \varphi_2(s_2) & \ldots & \varphi_p(s_2) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_1(s_r) & \varphi_2(s_r) & \ldots & \varphi_p(s_r) \end{bmatrix}. \tag{2.47}$$

the element kj-th of $\boldsymbol{\Phi}$ is j-th spline basis $\varphi_j(\cdot)$ evaluated at the k-th grid point s_k. The smoothing model can be also used in situations where different curves are measured at different grid points, in that case, different design matrices may be assigned to different curves. An example is provided in Section 4.3.

The smoothing model assumes that $\boldsymbol{\varepsilon}_i$ are Gaussian, uncorrelated, iid errors and therefore

observations $\boldsymbol{y}_i$ follow a conditional normal sampling distribution, that is

$$\boldsymbol{y}_i \mid \boldsymbol{\beta}_i,\ \tau_\varepsilon^2 \overset{\text{ind}}{\sim} N_r\left(\boldsymbol{\Phi}\boldsymbol{\beta}_i,\ \tau_\varepsilon^2 \boldsymbol{I}_r\right), \qquad \forall i = 1, \dots, n \tag{2.48}$$

where τ_ϵ^2 is the error precision and $\boldsymbol{I}_r$ is the $r \times r$ identity matrix.
The smoothing procedure (2.46) simply boils down to the estimation of the $\boldsymbol{\beta}_i$ coefficients, which are unknown and they have to be learned from data $\boldsymbol{Y}$. The simplest frequentist approach is to use a *least squares estimation* whose Bayesian counterpart is *linear regression model*. For this reason, we also refer to the $\boldsymbol{\beta}$s as regression coefficients.

We rely on a hierarchical Bayesian approach to smooth all functional observations simultaneously, enabling the borrowing of strength across all curves. To accomplish that, we place a prior distribution for the $\boldsymbol{\beta}_i$ coefficients given by

$$\boldsymbol{\beta}_1, \dots, \boldsymbol{\beta}_n \mid \boldsymbol{\mu}, \boldsymbol{K} \overset{\text{iid}}{\sim} N_p(\boldsymbol{\mu}, \boldsymbol{K}), \tag{2.49}$$

where $\boldsymbol{\mu}$ is the prior mean vector and $\boldsymbol{K}$ is the precision matrix. A traditional smoothing procedure would assume independence by setting $\boldsymbol{K} = \operatorname{diag}(\tau_1^2, \dots, \tau_p^2)$ while in this work we aim to improve the traditional smoothing by placing a Gaussian graphical model over the regression coefficients.

The B-spline basis expansion plays a key role in this extension: indeed each spline function has a compact support, which means that it is not null only in a defined portion of the domain. If the number of basis is large, the length of this each subinterval is very small if compared to the whole domain and therefore identifies a precise subinterval of the domain. The right panel of Figure 2.8 provides a visual representation of a 10-dimensional cubic B-spline basis. It also highlights that, as cubic splines are used, the support of each basis function overlaps with only four other functions supports. To facilitate the interpretation of the output of our model, we will consider each spline as representative of that subinterval where it greater than all the other ones.
Taking into account this hypothesis and the fact in (2.5) every spline φ_j is associated to a regression coefficient β_{ij}, for each $j = 1, \dots, p$, the part of the domain represented by each spline corresponds then to a particular slice of each curve, as displayed in the left panel of Figure 2.8. We can therefore associate the dependence structure of precise bands of the curve to the one describing the $\boldsymbol{\beta}$ coefficients. Finally, we anticipate that in the framework of spectrographic data analysis a physical interpretation is also available; each particular domain subinterval defines a slice of the spectrum that is associated to the absorbance property of the chemical mixture in a well defined frequency bands.

The way we aim to learn the above mentioned dependence structure is by framing the

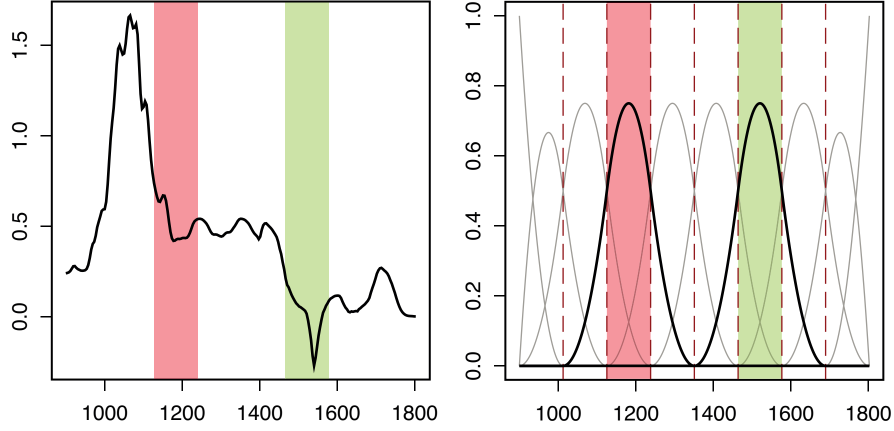

Figure 2.8. Example of a curve (left panel) and the corresponding 10-dimensional cubic B-spline basis (right panel).

prior distribution for the $\boldsymbol{\beta}$s into the Gaussian graphical modeling setting so that the forecited structure can be directly read off the estimated graph, as we explained in Chapter 1. The fundamental difference with respect to the previous chapter is that this time the structural learning is no longer performed on observed data but on the regression parameters, which are unknown and that have to be estimated as well.

We complete the model by placing vague, semi-conjugate, priors for the remaining parameters, ending up with the following Bayesian hierarchical model, here renamed as **Functional Graphical model**, or FGM for short.

$$
\begin{aligned}
\boldsymbol{y}_i \mid \boldsymbol{\beta}_i,\ \tau_\varepsilon^2 &\overset{\text{ind}}{\sim} N_r\left(\boldsymbol{\Phi}\boldsymbol{\beta}_i, \tau_\varepsilon^2 \boldsymbol{I}_r\right), \qquad \forall i = 1:n \\
\boldsymbol{\beta}_1,\ldots,\boldsymbol{\beta}_n \mid \boldsymbol{\mu},\ \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\boldsymbol{\mu}, \boldsymbol{K}\right) \\
\boldsymbol{\mu} &\sim N_p\left(\boldsymbol{0}, \sigma_\mu^{-2}\boldsymbol{I}_p\right) \\
\tau_\varepsilon^2 &\sim \text{Gamma}\left(a_\varepsilon, b_\varepsilon\right) \\
\boldsymbol{K} \mid G &\sim \text{GWishart}\left(b, D\right) \\
G &\sim \pi\left(G\right)
\end{aligned}
\tag{2.50}
$$

We highlight that in (2.50) and throughout this dissertation, we use an "Shape-Rate" parametrization for the Gamma distribution, which means that if $X \sim \text{Gamma}(a, b)$, its density function $f(x)$ is

$$
f(x) = \frac{b^a}{\Gamma(a)} x^{a-1} e^{-bx} \qquad \text{and its mean is} \qquad \mathbb{E}[X] = \frac{a}{b}
$$

Since the model exploits semi-conjugate priors, it is straightforward to derive all the full conditional distributions that are listed below.

- The full conditional of $\boldsymbol{\beta}$ coefficients is

$$\boldsymbol{\beta}_i \mid \text{all} \sim N_p\left(\boldsymbol{\mu}_i^*, \boldsymbol{K}^*\right) \qquad \forall i = 1 \ldots n \tag{2.51}$$

where $\boldsymbol{K}^* = \left(\tau_\varepsilon^2\ \Phi^T\Phi + \boldsymbol{K}\right)$ and $\boldsymbol{\mu}_i^* = (\boldsymbol{K}^*)^{-1}\left(\tau_\varepsilon^2\ \Phi^T\boldsymbol{Y}_i + \boldsymbol{K}\boldsymbol{\mu}\right)\ \forall i = 1 \ldots n$.

- The full conditional distribution of $\boldsymbol{\mu}$ parameters is

$$\boldsymbol{\mu} \mid \text{all} \sim N_p\left(\boldsymbol{m}^*, A^*\right) \tag{2.52}$$

where $A^* = \sigma_\mu^{-2}\boldsymbol{I}_p + n\boldsymbol{K}$ and $\boldsymbol{m}^* = (A^*)^{-1}\,\boldsymbol{K}\sum_{i=1}^n \boldsymbol{\beta}_i$.

- For τ_ε^2 we have

$$\tau_\varepsilon^2 \mid \text{all} \sim \text{Gamma}\left(a_\varepsilon\ \frac{nr}{2}, \frac{1}{2}\left(2b_\varepsilon + \sum_{i=1}^n \left(\boldsymbol{Y}_i - \Phi\boldsymbol{\beta}_i\right)^T\left(\boldsymbol{Y}_i - \Phi\boldsymbol{\beta}_i\right)\right)\right) \tag{2.53}$$

Regarding the graphical part of the model $(\boldsymbol{K}, \boldsymbol{G})$, observe that only the $\boldsymbol{\beta}$s are directly influenced, which allows us to rewrite the full conditional as

$$P(\boldsymbol{K}, G \mid \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n, \boldsymbol{\mu}, \tau_\varepsilon^2, \boldsymbol{Y}) = P(\boldsymbol{K}, G \mid \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n, \boldsymbol{\mu}) \tag{2.54}$$

- The full conditional for matrix $\boldsymbol{K}$ is then

$$\boldsymbol{K} \mid \text{all}\ \sim\ \text{GWishart}\left(b + n,\ D + \sum_{i=1}^n \left(\boldsymbol{\beta}_i - \boldsymbol{\mu}\right)\left(\boldsymbol{\beta}_i - \boldsymbol{\mu}\right)^T\right) \tag{2.55}$$

- Marginalizing the precision matrix out of the full conditional for the graph $\boldsymbol{G}$, we get that

$$P(G \mid \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n, \boldsymbol{\mu}) = \pi(G)\frac{I_G\left(b + n,\ D + \sum_{i=1}^n \left(\boldsymbol{\beta}_i - \boldsymbol{\mu}\right)\left(\boldsymbol{\beta}_i - \boldsymbol{\mu}\right)^T\right)}{I_G\left(b, D\right)} \tag{2.56}$$

Note how (2.55) and (2.56) do not explicitly depend on the data, which make them apparently different from (1.8) and (2.1). The reason is that in model (2.50) the graphical model is used to describe the $\boldsymbol{\beta}$ regression parameters, not the data. However the derived models are symmetric, indeed (2.54) clarifies that the relationship that bounds together $\boldsymbol{\beta}$, $\boldsymbol{\mu}$, $\boldsymbol{K}$ and $\boldsymbol{G}$ in model (2.50) is the same as model (1.9) where, intuitively and with some abuse of notation, the "data" are represented by $(\boldsymbol{\beta}_i - \boldsymbol{\mu})_{i=1:n}$. The way the graphical part is encapsulated inside the smoothing model should be clearly visible comparing, for example, equations (1.8) and (2.55), just remember that in Chapter 1 we assumed zero mean data.

Computationally speaking, the fact that $\boldsymbol{\beta}$s and $\boldsymbol{\mu}$ are not observable but they have to be estimated as well, poses some additional difficulties. We already discussed in Section 1.3.2 that in this case a Shotgun Stochastic Search is not applicable. Moreover, as the variables described by the graph lie on a lower hierarchy level with respect to the data, the information needed for its estimation are filtered, making the learning more challenging.

We exploit an **hybrid Gibbs sampling** strategy to simulate the posterior distribution of model (2.50). The regression parameters $\boldsymbol{\beta}$s, $\boldsymbol{\mu}$ and τ_{ε}^2 are drawn from their full conditionals and then an acceptance-rejection step is done for the graphical quantities $(\boldsymbol{K}, \boldsymbol{G})$. In particular we use a Block Double Reversible Jumps method, see Algorithm 11, if we want to impose a block structure on the graph or a Double Reversible Jumps one, see Algorithm 10, if we want to search over the entire space of all possible graphs. Further computational details are given in Chapter 3 where the most interesting coding technique used in the `BGSL` package are presented.

Chapter 3

BGSL Package

We now present all computational aspects of `BGSL`. The code was written completely from scratch. As always in those cases, this implied a lot of extra work but it also gave us total freedom of shaping it according to our philosophy. Here is a list of some key ideas which the code is based on. They are a way of having a standardized style all over the code and we believe that keep pushing in that direction would blaze a trail for enormous improvements. Even though new ideas are always well welcomed, we warmly encourage future developers to try to follow them as much as possible.

- **Independence with respect to software interface:**
  `BGSL` is an `R` package but it is build in such a way that it could be totally independent from it. We choose `R` because at present day it is the main reference software for the statistical community, especially in academia. Of course, it is not the only choice, Python is rapidly gaining popularity and could become the standard in a few years.

  We then decided to write `R` depending code only on the higher possible level, that is only for handling inputs and outputs. Indeed, `BGSL` data structures, functions and samplers are written exclusively in C++. It is actually possible to compile and run everything within a pure C++ environment. This makes it really easy to extract the core code and adapt it to a new framework. Only the front end functions have to be rewritten. To be precise, only the `BGSL_export.cpp` and `BGSL_export.R` files. Of course, this choice has some drawbacks.

  Rcpp (Eddelbuettel and François 2011) is the main tool to enable C++ programming within an R environment; it does more then simply compiling C++ functions and making then available as `R` functions. For example, it allows the usage of `R` code blocks within C++ functions and it also provides some types meant to automatically convert `R` types into C++ types, enabling a semantic very similar to what would actually be written in `R`. Rcpp basic linear algebra is then incredibly flexible but for sure it is not as powerful as the one provided by other C++ libraries. Nevertheless, we belive that the possibility

of having code that can be easily interfaced with other software is more important than integrating it with R function calls.

- **Code generality:**
  BGSL relies considerably on the use of templates. This choice was taken mainly for two reasons. First, the code was developed completely from scratch, which implies that many different data structures were tested and changed many times. Having a general code helped a lot. The second and most important reason is that the whole package is built around the duality between block and complete graphs. We then decided to templatize all graph-related functions with respect to the type of graph they operate on. This choice led the way to other function-specific templatizations which make the code very flexible. However, it is not, or maybe not yet, a template only library, in particular classes that interface with extern libraries are not.

  The choice of many template functions and classes was made more for generality and flexibility rather than for optimization. Nevertheless, once that template structure is correctly set, it is the skeleton to allow new improvement via template metaprogramming techniques.

- **Facilitate future extensions:**
  BGSL would like to be a starting point for other possible works on Gaussian graphical models. A GGM is identified by three components. The type of search algorithm one wants to use, the choice of the prior for the graph and the choice of prior for the precision matrix. Our code forces this last choice to be a GWishart prior but allows for extensions of the other two. This is possible because **GraphPrior.h** and **GGM.h** implement polymorphic classes, each having its own generic factory for selecting the desired polymorphic object. One just needs to add a new prior or a new GGM search method and it would be automatically available in all samplers.

  Moreover, the best way for boosting the library is to improve the graph classes. They are indeed zero-level data structures that are repeatedly called throughout the code. Our choice can be bettered and new implementations are more than welcomed. The problem is that graph objects are used all over the classes and, if a new type were provided, it would be hard to add it in every function or class that uses graphs. This is another reason why we choose to use templates. Graph types are always template parameters. Therefore, if a new graph type is implemented, the only thing to do is to let it know to **internal_type_traits.h** and it would be available everywhere. More details are given in Section 3.2.2.

It is moreover important to underline that computational aspects related to Bayesian statistics are particularly insidious. The goal is usually to create an efficient sampler. The difficulty

is that this is usually made of many simple pieces that are repeated many many times. The consequence is that, to optimize the code, one has to work around very simple operations that usually look fine the way they are.
Finally, the ultimate challenge is how to handle memory. By definition, the output of a Monte Carlo Markov chain is a very long sequence of data. If it is too long or if the sampled objects are too big, memory handling becomes an issue. This problem is deeply analyzed in Section 3.1.3.

We now proceed presenting the actual implementation of the package. We start showing what external libraries were used and how they were wrapped to fit in our code.

3.1 External C++ libraries

BGSL depends on three C/C++ external libraries. The first one is the **Eigen** library. It is a template only library for linear algebra. It is extensively used throughout the code. Matrices, vectors, decomposition algorithms such as Cholesky factorisation and solvers are taken from the **Eigen**. We do not present here how it was exploited, several examples will be provided in the following sections. As **Eigen** types may result cumbersome to be written, we here report in Code 3.1 some useful typedefs used in BGSL and extensively used in all following examples.

```
//Eigen:: is omitted
using MatRow    = Matrix<double, Dynamic, Dynamic, RowMajor>;
using MatCol    = MatrixXd;
using MatUnsRow = Matrix<unsigned int, Dynamic, Dynamic, RowMajor>;
using MatUnsCol = Matrix<unsigned int, Dynamic, Dynamic, ColMajor>;
using VecCol    = VectorXd;
using VecRow    = RowVectorXd;
```

Code 3.1. Typedefs for some Eigen types

The second external library is the **GNU Scientific Library** or simply *GSL*. It is a numerical library that covers a lot of different areas of scientific programming in a very efficient way. It is used for two purposes, the creation of smoothing basis splines (B-splines) and for random number generation, used to sample from simple, well-known random number distributions. About that, its performances were tested and compared with **STL** and **Boost** corresponding utilities. It turned out that **GSL** outperforms both of them. The drawback is that it is a C library which relies on row pointers and dynamic memory allocation via a mechanism that follows the style of `malloc` and `free`. As sampling from random distributions is the building block of Bayesian statistics, we decided to wrap **GSL** C-style functions into C++ classes. This allows for simpler and safer coding in all other classes. This is done in the `sample` and `spline` namespaces.

3.1.1 The sample namespace

First ingredient for random number generation is of course a random number generator. All **GSL** sampling functions have a firm that looks like this one,
`double gsl_ran_gaussian(const gsl_rng * r, double sigma)`.
The first parameter is a **GSL** random number generator. This means that it is not possible to use generators from other library.
The `GSL_RNG` class wraps `gsl_rng`. A synthesized example is provided below

```
class GSL_RNG{
    public:
        GSL_RNG(unsigned int const & _seed){ //get seeding value
            if(_seed == 0){ //generate random seed
                seed = static_cast<unsigned int>
                       (std::chrono::steady_clock::now().
                       time_since_epoch().count());
                std::seed_seq seq = {seed}; //add entropy
                std::vector<unsigned int> seeds(1);
                seq.generate(seeds.begin(), seeds.end());
                seed = seeds[0];
            }
            else{
                seed = _seed;
            }
            gsl_rng_env_setup(); //set gsl environment variables
            r = gsl_rng_alloc(gsl_rng_default); //allocate memory
            gsl_rng_set(r,seed); //seed the engine
        }
        ~GSL_RNG(){
            gsl_rng_free(r); //deallocate memory
        }
        gsl_rng* operator()()const{
            return r;
        }

    private:
        gsl_rng * r;
        unsigned int seed;
};
```

Code 3.2. GSL Random number generator wrapper

`gsl_rng_env_setup()` function reads the environment variable `GSL_RNG_TYPE` and uses its value to set the corresponding library variable `gsl_rng_default` which defines the `mt19937` engine. It is not the only possible choice but it is more than fine. Another engine, for example `taus`, could be used by simply calling `gsl_rng_taus`. A full list of possible engines can be

found here https://www.gnu.org/software/gsl/doc/html/rng.html. The selected engine is then seeded using `gsl_rng_set(r,seed)` function.

`BGSL` notation establishes that if the seed is null, then it has to be randomly generated. This step is delicate because of compatibility problems with Windows operating system. The best way to achieve this task is indeed to use `std::random_device`. The problem is that not all architectures provide a random device. In particular, there is a bug in the compiler provided within the MinGW environment, that is the one used by `R` when running on Windows, see here. It was actually solved starting from version GCC 9.2 but the one provided by Rtools is still version 8.3 and we have to keep that one. More details on the topic will be given later. The trick that solves the problem is to generate the seed using `std::chrono::steady_clock` in combination with `std::seed_seq`. The latter is needed since `std::chrono::steady_clock` does not guarantee sufficient entropy in close, subsequent calls.

The `gsl_rng *` that is needed for function calls is then accesible by means of the call operator. Finally, as this object is destroyed, it calls `gsl_rng_free(r)` which deallocates the memory according to **GSL** standards.

Once that the random number generator is set, it is used as an underlying source of randomness for drawing samples from some random distributions. Random variates needed by `BGSL` can be distinguished into two groups. Univariate random variables, such as uniform, integer uniform, normal, gamma and chi-squared and multivariate random variables such as normal multivariate and the Wishart distribution. We decided to implement them as callable objects, following the **STL** style.

For what concern the first group, there are no special considerations to do. We simply wraps **GSL** functions. Here is a brief example.

```
struct runif
{
    double operator()(GSL_RNG const & engine)const{
        return gsl_rng_uniform(engine());
    }
};
//This function returns a random integer from 0 to N-1
struct runif_int
{
    unsigned int operator()(GSL_RNG const& engine, unsigned int const& N)const{
        return gsl_rng_uniform_int(engine(), N);
    }
};

struct rnorm
{
    //Get standard deviation
    double operator()(GSL_RNG const& engine, double const& mu, double const& sd)
```

```
    const{
            return gsl_ran_gaussian_ziggurat(engine(),sd) + mu;
    }
};

struct rgamma{
    double operator()(GSL_RNG const & engine, double const & shape, double const &
    scale)const{
        return gsl_ran_gamma(engine(),shape,scale);
    }
};

struct rchisq{
    double operator()(GSL_RNG const & engine, double const & k)const{
        return gsl_ran_chisq(engine(),k);
    }
};
```

Code 3.3. GSL univariate distributions wrapper

The only point worth mentioning is to highlight that `gamma` follows a "Shape-Scale" parametrization, that is

$$\text{if } X|\text{shape}, \text{scale} \ \sim \text{Gamma}(\text{shape}, \ \text{scale}), \text{ then } \ \mathbb{E}[X] = \text{shape} * \text{scale}$$

Sampling from multivariate distributions is instead much more interesting. The goal is to draw a sample from $N_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ following a covariance parametrization and from Wishart$(b, \boldsymbol{\Psi})$. Little care should be given to this parametrization. $\boldsymbol{\Psi}$ is a $p \times p$ Scale matrix and b is the Shape parameter that is not equal to the degree of freedom. This is done so that Wishart and GWishart are both parametrized with respect to Shape parameter . To avoid any possible confusion, we here report the Wishart density and its expected value,

$$f(\boldsymbol{X}) = \frac{|\boldsymbol{X}|^{(b-2)/2} \ \exp\left\{-\frac{1}{2}tr\left(\boldsymbol{X} \ \boldsymbol{\Psi}^{-1}\right)\right\}}{2^{p(b+p-1)/2} \ |\boldsymbol{\Psi}|^{(b+p-1)/2} \ \Gamma_p((b+p-1)/2)}, \ \ \mathbb{E}[\boldsymbol{X}] = (b+p-1) \ \boldsymbol{\Psi}$$

where $\Gamma_p(m)$ is the multivariate gamma distribution, defined, for $m > (p-1)/2$, as

$$\Gamma_p(m) = \pi^{p(p-1)/4} \prod_{i=0}^{p-1} \Gamma\left(m - \frac{i}{2}\right)$$

Both sampling algorithms need to perform a Cholesky decomposition of matrices $\boldsymbol{\Sigma}$ and $\boldsymbol{\Psi}$. However, it often happens in the code that such factorizations are available in advance, and this is something that can be exploited. We here show some insides for what concern the `sample::rwish` class, the same considerations apply also to `sample::rmvnorm`.

```
enum class isChol{ Upper, Lower, False };

template<typename RetType = MatCol, isChol isCholType = isChol::False>
struct rwish{
    template<typename Derived>
    RetType
    operator()(sample::GSL_RNG const& engine, double const& b,
               Eigen::MatrixBase<Derived>& Psi )const
    {
        static_assert(isCholType == isChol::False ||
                      isCholType == isChol::Upper ||
                      isCholType == isChol::Lower ,
                      "Error, invalid sample::isChol field inserted");
        //GSL matrices needed
        gsl_matrix *cholMat = gsl_matrix_calloc(Psi.rows(), Psi.rows());
        gsl_matrix *work    = gsl_matrix_calloc(Psi.rows(), Psi.rows());
        //Create return object
        RetType return_obj(RetType::Zero(Psi.rows(), Psi.cols()));
        gsl_matrix result;//gsl_matrix where the sampled values are stored.
        result.owner  = 0;  //result does not own the buffer of stored data
        result.data   = return_obj.data();  //bind the buffers

        if constexpr(isCholType == isChol::Upper){
            if constexpr( Psi.IsRowMajor ){
                /* operations and more if constexpr clauses ...*/ }
        }

       //Sample with GSL
       gsl_ran_wishart(engine(), b+Psi.rows()-1, cholMat, &result, work);

       //Free memory and return
       gsl_matrix_free(cholMat);
       gsl_matrix_free(work);
       //result is no freed because no extra memory was allocated
       return return_obj;
    }
```

Code 3.4. GSL multivariate distributions wrapper

Those classes are then templatized with respect to the type of Cholesky decomposition that can be set by means of `isChol` enum class, which is checked using a `static_assert`. The call operator itself is a template, so that it is possible to use it both with RowMajor and ColumnMajor **Eigen** matrices. Inside the body of the function, some preliminary operations have to be performed according to what combination of `isCholType` and `EigenType` is passed. This is carefully and efficiently achieved using different `if constexpr()` clauses.
There is then a second problem. This functions are wrapping existing functions that are

written to be used with **GSL** matrix and vector types. There are several reasons why we instead prefer working with **Eigen** objects. The simplest way to proceed is to get as input an **Eigen** type, copy it into an **GSL** one, compute the result and then copy it back. This way actually implies two useless copies. Since `sample::rwish` is mainly used within the `utils::rgwish()` function, that is the most important call of every `BGSL` sampler, we would prefer to avoid those copies. In practice, this is done by creating a **GSL** matrix does not own its own buffer but uses directly the one of the **Eigen** object as shown in Code 3.4. In that case, writing on `result.data` is like writing on `return_obj.data()` and viceversa. Moreover, setting `result.owner` to 0 implies that when that object is destroyed, the buffer remains untouched so that we are able to correctly return the sampled matrix. This is for sure a dangerous operation, but testing it with Valgrind memcheck, no possible leaks were found.

The last object belonging to this namespace is `sample::rmvnorm_prec` that is used to sample from the multivariate normal distribution according to precision parametrization. It is indeed possible to construct a sampling strategy such that it is possible to avoid inverting the whole precision. We do not provide here other details because there is neither an interesting usage of **GSL** library nor particular optimizations applied. It is not indeed used inside the sampler, but it is useful for data simulation and completeness of the class. Further details can be directly found in `GSL_wrappers.h` file.

3.1.2 The `spline` namespace

The following namespace is a collection of functions wrapping **GSL** utilities for computing smoothing basis splines, B-splines for short. As the name suggests, they are commonly used as basis functions to fit smoothing curves.
To define them on the interval $[l, u]$, the abscissa axis is broken up into some number of subintervals, where the endpoints of each interval are called *breakpoints*. The number of breakpoints is called *nbreak*. These are then converted to knots by imposing various continuity and smoothness conditions at each interface. Note that the boundary points are always part of this set, the number of internal knots is then *nbreak* -2. Given a sequence of $p+k$ nondecreasing breakpoints

$$t = \{t_0, t_1, \dots, t_{n+k}\}$$

the p basis splines of order k are defined by the following recursive relationship

$$\begin{aligned}
\varphi_{i,1}(s) &= \mathbb{1}_{[t_i, t_{i+1})}(s) \\
\varphi_{i,k}(s) &= \frac{(s - t_i)}{(t_{i+k-1} - t_i)}\varphi_{i,k-1}(s) + \frac{(t_{i+k} - s)}{(t_{i+k} - t_{i+1})}\varphi_{i+1,k-1}(s)
\end{aligned}$$

for $i = 0, ..., p-1$. The above recurrence relation can be evaluated in a numerically stable way by the de Boor algorithm, which is what is implemented in **GSL** function `gsl_bspline_workspace * gsl_bspline_alloc(const size_t k, const size_t nbreak)`. *nbreak*, p and k are not independent but they are defined by the relationship $p = nbreak + k - 2$. Given an appropriate set of knots on an interval $[l, u]$, then the B-spline basis functions form a complete set on that interval. A standard choice for those knots is to choose them uniformly spaced, that is also the case for `BGSL`. Therefore, we can expand a smoothing function as

$$y(s) \approx \sum_{i=1}^{p} \beta_i \varphi_i(s)$$

where the $\boldsymbol{\beta}$ coefficients are unknown and have to be estimated.

`BGSL` functional samplers do not actually need the B-spline basis function in their continuous form, they only need their evaluation of a particular grid of r points. Do not confuse the *nbreak* breakpoints with the r grid points. The first ones are the one needed to define the B-spline basis function, the second ones are the one needed for the estimation of regression coefficients $\boldsymbol{\beta}$.

We set $\boldsymbol{\Phi}$ to be the B-spline design matrix. $\boldsymbol{\Phi} \in \mathbb{R}^{r \times p}$ defined as follows:

$$\boldsymbol{\Phi} = \begin{bmatrix} \varphi_1(s_1) & \varphi_2(s_1) & \dots & \varphi_p(s_1) \\ \varphi_1(s_2) & \varphi_2(s_2) & \dots & \varphi_p(s_2) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_1(s_r) & \varphi_2(s_r) & \dots & \varphi_p(s_r) \end{bmatrix}$$

the i-th row contains the evaluation of all p basis functions over the same grid point s_i while the j-th column contains the evaluation of the j-th spline basis over the whole grid.

The `sample` namespace defines the following functions

```
tuple<MatCol, vector<double> >
generate_design_matrix(unsigned int const& order,
                       unsigned int const & n_basis, double const & a,
                       double const & b,
                       vector<double> const & grid_points);
//Same as before but assumes that the grid points are r uniformly space
//points in [a,b]
tuple<MatCol, vector<double> >
generate_design_matrix(unsigned int const & order,
                       unsigned int const & n_basis, double const & l,
                       double const & u, unsigned int const & r);
//Computate derivatives up to order n_deriv
std::vector<MatCol>
evaluate_spline_derivative(unsigned int const & order,
                           unsigned int const & n_basis, double const & l,
```

```
                           double const & u,
                           vector<double> const & grid_points,
                           unsigned int const & nderiv);
//Same as before but assumes that the grid points are r uniformly space
//points in [a,b]
vector<MatCol>
evaluate_spline_derivative(unsigned int const & order,
                           unsigned int const & n_basis, double const & l,
                           double const & u, unsigned int const & r,
                           unsigned int const & nderiv);
```

Code 3.5. GSL spline wrapper

The first two functions of Code 3.5 returns $\mathbf{\Phi}$ and a vector containing the internal knots that are used to define the spline basis. The difference between the two is that the second one assumes and constructs r uniformly spaced grid points in $[l, u]$, while the first one needs to have them explicitly passed. The remaining functions return not only $\mathbf{\Phi}$ matrix but also the evaluation of spline derivatives up to some desired order. We use the notation such that order zero is $\mathbf{\Phi}$ itself. Their implementation does not contain any particular optimization technique as the $\mathbf{\Phi}$ matrix has to be constructed only once. Nevertheless, they are a good example of how to use **GSL** objects. Implementation details can be found in `GSL_wrappers.cpp` file.

3.1.3 The `HDF5conversion` namespace

The third and last external library used in `BGSL` is HDF5. As mentioned above, saving the drawn samples from a MCMC may rapidly become an issue. People with experience in this field are for sure aware of problems such as program crashing or `R` session aborted.

`BGSL` solution for those problems is actually not to store any data in memory. We choose to write directly on a file. However, standard I/O would slow down the code too much. That is why we decided to use directly binary I/O. It consists in writing and reading directly from or to a buffer, i.e a contiguous memory area. This type of operation is much faster than the standard one but, among other problems, it does not generate a human-readable file. Moreover, that binary files usually are architecture-dependent, which is of course a limitation for what concern the portability of the package.
The **HDF5** library allows instead to create not only architecture-independent binary files, but also with named "subfolders", which helps a lot the comprehension and organization of data. A full description of this library can be found in their reference manual. We here give some details on how we used it for our purposes. Note that we can not be completely satisfied by this implementation as it is very basic and not general at all, which goes against one of our style rules.

The main ingredients for using **HDF5** library are:

- **A file:** It is created via `H5Fcreate()` function which takes as input the name to be given to the file and the access mode. Possibilities are `H5F_ACC_EXCL` that generates an error if there exists already a file with that specified name or `H5F_ACC_TRUNC` that instead creates a new file in any case, even deleting an existing one. To open a file instead one has to use the `H5Fopen()` function which also takes the file name and an access mode that can be read only `H5F_ACC_RDONLY` or `H5F_ACC_RDWR` read and write. Files are not used to directly write on them, they are a way for collecting several *dataset* that are the objects where one actually writes on. Before introducing them, we first need to present their building block, *dataspaces*.

- **Dataspaces:** They are used to define the *dataset* dimension. The first thing to specify is the `rank`, that is the dimensionality of the desired dataset. If one, we have a linear dataset, if two a bi-dimensional object such as a matrix, three for a 3-dimensional matrix and so on. Then we need to declare the size for each of those dimensions. Once we have done that, it can be created using `H5Screate_simple()`.

- **Datasets:** They are the most important object, the one that is actually use for reading and writing. They are defined given a file, a type and a dataspace. For what concern the type there are several possibilities, we decided simply to use **HDF5** native type such as `H5T_NATIVE_UINT`. Moreover, it is possible to give a name to each dataset. In this way, a single file can have different named datasets that work exactly like subfolders. Writing is done by using the following function:
  `H5Dwrite(dataset,mem_type,mem_space,file_space,xfer_plist,buf )`.
  The first argument is the dataset where `buf` has to be written. The second is used to specify the type of elements in the buffer. `mem_space` and `file_space` parameters are meant to describe the dataspaces in memory, i.e the object we want to write down, and in the file, respectively. They can be set to the value `H5S_ALL` to indicate that an entire dataset is to be written. That is not the only option, in a later section we look at how we access only a specific portion of a dataset. Finally, the fifth element sets some properties, but it is usually simply defaulted using `H5P_DEFAULT`.
  Reading is done similarly by means of `H5Dread()`.

- **Hyperslabs:** As anticipated in the previous section, we are not forced to read and write the whole dataset, it is actually possible to access only a part of it. This is done thanks to *hyperslabs selection*. As the name suggests, it consists in indicating which subsection of the dataset we would like to access. It is done using function `H5Sselect_hyperslab()`. Its usage is not so intuitive, and that is because it actually allows for selecting not only a contiguous sub-region but even overlapping parts of the dataset or a recurrent pattern. We are actually interested only in the simplest possible case, so we do not

provide here a complete description of the function, which may just result confusing, but we directly show its usage in the next coding example. Figure 3.1 shows some examples of its potential.

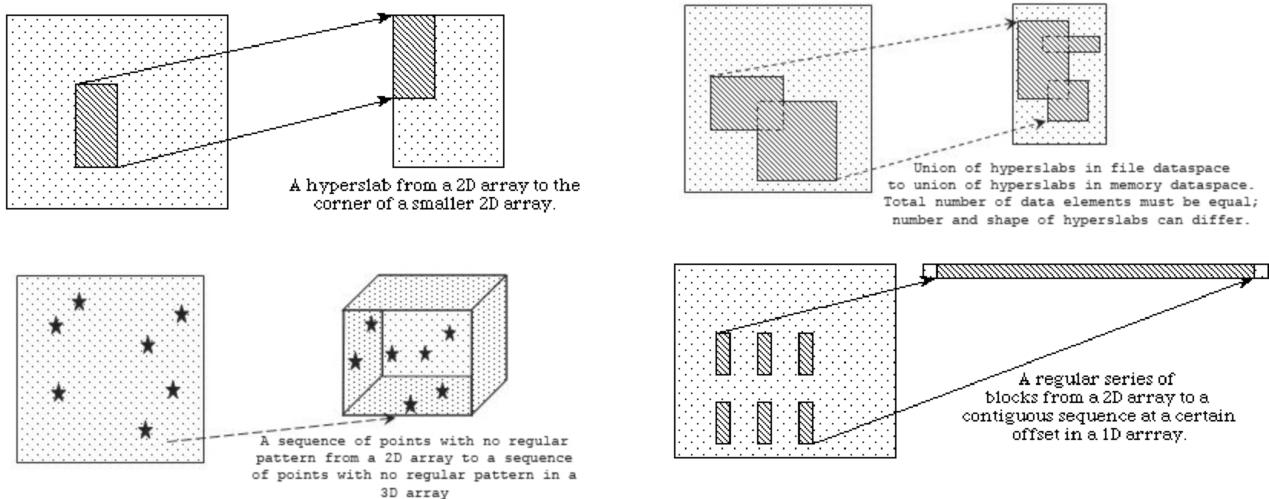

Figure 3.1. Images taken from HDF5 user's guide representing possible hyperslab selections and points selection.

- **Points selection:** Hyperslabs are meant to select regular subsets of a dataset. If one wants to select just a list of points, possibly with no regular pattern, the function to be used is `H5Sselect_elements()` which takes as input an array containing the coordinates of the desired points. Figure 3.1 provide an example of points selection.

That was an simple overview of the most important **HDF5** functions that we used.

As already mentioned, the final goal is to write the states visited by the MCMC directly on a binary file instead of storing them in memory. All `BGSL` samplers generate an output file. `BGSL` provides four different sampler, all described in Section 3.5. For the moment, we limit ourselves to present the structure of the file generated by `FGMsampler`, the one that samples from model (2.50). It is indeed the most complete one, the others can be seen as subcases of this one, at least in terms of what variables come into play. The main differences in the output of `GGMsampler` and `FLMsampler` are briefly discussed in their respective presentations, see Section 3.5.

- **File:** Each sampler has as output a HDF file. Its name can be chosen by the user or just defaulted. It is important to remember that files are created using `H5F_ACC_TRUNC` access mode, that is the one that deletes an existing file having the same name. We then strongly recommend not to use defaulted names because it could lead to the loss of the previous one. The inserted name should not contain the extension suffix, that is automatically added and it is `".h5"`.

- **/Info:** Each file has a dataset called "/Info" storing four information: p that is the number of vertexes of the graph in its complete form. n, the number of observed data, stored_iter, the number of saved iterations for all those quantities that are not related to the graphical part, i.e "Beta", "Mu", "TauEps", and stored_iterG that is the number of saved iterations for the graphical part, that are of course "Precision" and "Graphs". Those four quantities are needed for reading datasets containing the sampled values. They are of course information that should be available to the user even without reading them from the file, the problem is that, as it is not in human readable form, if one forgets those parameters then it becomes impossible to read the file. They are a sort of policy that ensures that data can always be extracted from it.
  Those information can be read from the file using `HDF5conversion::GetInfo()`, the only input is the name of the file, extension included.

- **/Sampler:** This dataset simply saves the name of the sampler that was used. Its purpose is to simplify the reading of the file and to prevent wrong requests.

- **/Beta:** This is a bi-dimensional dataset of size $p\times$stored_iter$*n$. At each iteration, $p\times n$ regression coefficients are drawn and are written on file one after the other. They represent variables $(\boldsymbol{\beta_1}, ..., \boldsymbol{\beta_n})$ of model (2.50). The new matrix can be written on the dataset using `HDF5conversion::AddMatrix()` and read using `HDF5conversion::ReadMatrix()`. Moreover, for posterior analysis, it is important to extract the whole chain of sampled values of a single β_{ij} coefficient, the i-th regression coefficient of the j-th curve. To do that, use `HDF5conversion::GetChain_from_Matrix()`.
  The example below provides more details about their usage.

- **/Mu:** This is a one-dimensional dataset of length $p * stored_iter$. At each iteration, p coefficients are drawn and are written on file one after the other. They represent variable $\boldsymbol{\mu}$ of model (2.50).
  Writing and reading from and to this dataset can be done using `HDF5conversion::AddVector()` and `HDF5conversion::ReadVector()`. Moreover, the extraction of the chain for element μ_j is done with `HDF5conversion::GetChain_from_Vector()`.

- **/Precision:** This is a one-dimensional dataset of length $\frac{p(p+1)}{2} stored_iterG$. At each iteration, a $p \times p$ symmetric random matrix is drawn. Because of symmetry, it is actually useless to save the lower triangular part. We underline that the elements are written row by row, diagonal included. They represent the variable $\boldsymbol{K}$ of models (1.9) and (2.50).

- **/TauEps:** This is a one-dimensional dataset of length $stored_iter$. At each iteration, one single elements is drawn and written on file after the previously sampled one. It repre-

sents variable τ_ε^2 of model (2.50). Writing, reading and chain extraction can be done as for "/Mu", the only difference would be the size of the provided buffer. Dataset operations are performed via `HDF5conversion::AddVector()`and `HDF5conversion::ReadScalar()`. In this case there is no need for a chain extraction because this dataset is itself the chain for τ_ε^2, as it is the only scalar quantity.

- **/Graphs:** The last dataset is the one for graphs, used in models (1.9) and (2.50). As for precision matrix, only the upper triangular part of the matrix is saved row by row. Its size depends on what type of graphical model is used. For complete graphs, there is no need of saving the diagonal elements, so its length is $\frac{p(p-1)}{2} stored_iterG$. Block graphs instead may require to have also the diagonal saved, it actually depends on the number of *singletons* in the groups. The general length is $\frac{p(p-1)+p-\#\text{singleton}}{2} stored_iterG$. Use `HDF5conversion::AddUintVector()` and `HDF5conversion::ReadUintVector()` for writing and reading. For what concerns posterior extraction of data, we provide an utility,
`HDF5conversion::GetGraphsChain()` which creates an `std::map` containing all visited graphs and their absolute frequencies. It also returns the chain of graph sizes, a useful tool in convergence analysis.

It is easy to see that the implemented functions can be divided in three groups, operating on matrices, on vectors or scalars. We are now describing some coding details regarding the matrix case. It is indeed the most complete example we can provide, and all other cases are simple simplifications of this one. Further details can be found in `HDF5conversion.h` and `HDF5conversion.cpp` files

```
/*Takes a bi-dimensional dataset of dimension (p x iter_to_store*n) and adds
a column matrix (p x n) starting from position iter*/
void
AddMatrix(DatasetType & dataset, MatCol & Mat, unsigned int const & iter)
{
    ScalarType p = static_cast<ScalarType>(Mat.rows());
    ScalarType n = static_cast<ScalarType>(Mat.cols());
    MatRow Mat2(Mat); //hdf5 stores matrices in C-style that is row-wise.

    double * buffer = Mat2.data();  //Get buffer
    ScalarType offset[2] = {0,iter*n};//initial point in complete matrix
    ScalarType count [2] = {1,1};//Means that only one block is added.
    ScalarType stride[2] = {1,1};//Means that only one block is added.
    ScalarType block [2] = {p,n};//sub-matrix dimensions

    DataspaceType dataspace_sub;
    dataspace_sub = H5Dget_space(dataset); // get dataspace
```

```
    //describe the "selection" for that dataspace
    StatusType status = H5Sselect_hyperslab(dataspace_sub, H5S_SELECT_SET,
                    offset, stride, count, block);

    //Describe the dataspace for the object in memory.
    MemspaceType memspace;
    int rank_mem = 1; //linear buffer
    ScalarType dims_sub = p*n; //same number of elements of the matrix
    memspace = H5Screate_simple(rank_mem, &dims_sub, NULL);

     //Write
     status = H5Dwrite(dataset, H5T_NATIVE_DOUBLE, memspace, dataspace_sub,
                H5P_DEFAULT, buffer);
}
```

Code 3.6. HDF5conversion

Code 3.6 is a short but exhaustive example of all the functions from **HDF5** we used. Given a dataset, we select its dataspace with `H5Dget_space()` and define a selection of the latter using `H5Sselect_hyperslab()`. Beside the dataspace, this function wants four more parameters as input, `offset, count, stride, block`. We actually need only two of them; the first one states what is the starting point of this selection; in our case the new matrix is always inserted starting from the first row and every n columns. We are indeed adding $p \times n$ matrices each iteration, as it is specified by last parameter, `block`. The other two inputs are for more advance usage, like the one that are shown in Figure 3.1.
The hyperslab selection of the dataset of the dataspace is not enough. We also need to specify a dataspace for the object in memory we want to write, i.e the matrix. This is needed because, in general, it could be possible to write only a part of that matrix, which is not needed in our case. The only important requirement is that it has to have the same number of selected elements of the one previously defined. It does not have to be bi-dimensional, and, for the sake of simplicity, that is why we created a linear selection. Once it is done, it is straightforward to write on that dataset.

3.2 Basic graphical objects implementation

In this section, we describe the zero level classes, the ones defining our most basic objects that are "Groups", "Graphs" and "Precision" matrix. Block graphs are defined only with respect to some groups and the precision matrix, since it has to be distributed according to GWishart distribution, it can be defined only with respect to some graph.

3.2.1 Groups

Declarations and implementations of the `Groups` class can be found in `Groups.h` and `Groups.cpp` files. The purpose of this class is not just to define what the groups are but also to mimic the behaviour ρ and ρ^{-1} functions, defined in (1.11) in Section 1.1.
Let us consider the following reference example,

$$
\begin{aligned}
0 &\rightarrow \{0,1,2,3\} \\
1 &\rightarrow \{4\} \\
2 &\rightarrow \{5,6,7\} \\
3 &\rightarrow \{8,9\}
\end{aligned}
\tag{3.1}
$$

We defined four groups of different lengths. As in Chapter 1, we refer to the number of groups by simply calling it M. The first group, the length of the one indexed by 0 is equal to three, the second group is just a *singleton* since it contains only one element and so one. It describes the mapping of a four-block node graph into a complete graph having ten vertices. Enumeration of nodes is zero-based, following a C-style convention. Their definition has to follow some rules,

- Every set of grouped nodes has to be ordered. It is not possible, for example, to exchange 5 and 6 in the third group.
- As stated in (1.10), Section 1.1, all indexes that belong to the k-th group have to be strictly larger than the indexes in all previous groups. In our example, it would not be possible to exchange index 5 in group 2 with index 1 in group 0.
- All p nodes must belong to some group, eventually as singletons.

The `Group` class definition looks like this,

```
class Groups : public GroupsTraits,
                      std::vector<std::vector<unsigned int>>
{
  std::vector<unsigned int> map_of_indeces;
};
```

Code 3.7. Groups class

It is defined by publicly inheriting from `std::vector<std::vector<unsigned int> >` which is useful in order to immediately inherit all nice properties of the well-known **STL** class. In this way, the `Groups` class itself is a sort of representation of the ρ function. By simply accessing `(*this)[i]` element we can read all those nodes of the complete graph that are related to the i-th node of the graph in block form, just like in the previous example.

The `map_of_indeces` public member is instead meant to represent the inverse operation. It is a vector of length p such that the i-th element is equal to the group which the i-th vertex of the complete graph belongs to. It is exactly the sort of information that is carried by ρ^{-1}. Taking as reference example (3.1), the corresponding `map_of_indeces` would be

0	0	0	0	1	2	2	2	3	3

3.2.2 Graphs

Graphs are divided into two different categories, "Complete" and "Block" graphs. We start introducing the first one.

- A complete graph $G = (V, E)$ is an undirected graph, where $V = \{1, ..., p\}$ is the set of nodes and E is the edge set. It can be seen as a point in the space of all possible graphs having p vertices, the one we called $\mathcal{G}$, see Chapter 1. Its algebraic representation is given by the adjacency matrix, a $p \times p$ symmetric matrix such that entry (i, j) is equal to 1 if edge $(i, j) \in E$, 0 otherwise. It not is actually the only possible representation for a graph but it is the one most suitable for our needs.
  `BGSL` type which represents such a graph is `GraphType`, whose implementation is available in the file `GraphType.h`. Its definition is the following,

```
template<class T=unsigned int>
class GraphType : public GraphTypeTraits<T>{
public:
  /* Other pulic methods are available */
  inline GraphType& completeview(){
    return (*this);
  }
  // Call operators
  T operator()(IdxType const & i, IdxType const & j)const{
    return (i<=j) ? (data(i,j)) : (data(j,i));
  }
  T& operator()(IdxType const & i, IdxType const & j){
    return (i<=j) ? (data(i,j)) : (data(j,i));
  }
private:
 Matrix<T,Dynamic,Dynamic,RowMajor> data;
 map<unsigned int, vector<unsigned int> > neighbours;
};
```

Code 3.8. GraphType class

`data` is where the adjacency matrix is stored. Its type is `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>`, the reason why it is stored by rows will become clear later on. It is a class template, the admissible types are `bool`, `unsigned int`, `int` but there are some situations when even a `double` could be useful, for example when dealing with *plinks* matrix, which is a significant summary of the sampled graphs that is described in Section 3.5.4. An adjacency matrix is symmetric by definition, but, for efficiency reasons, there are no **Eigen** types that allow to store only the upper triangular part of the matrix. We specialized the call operator so that we are sure that only the upper triangular part could be referenced.
We also save a second member, called `neighbours`. Let us first provide some definitions. In graph theory for undirected graphs, the neighbourhood of vertex i is defined as

$$N_i = \{j \in V \mid (i, j) \in E\} \tag{3.2}$$

that is the set of all those vertices that are directly connected to the i-th node. In the `neighbours` member where are storing this kind of information for every possible node.

- A block graph $G_A = (V_A, E_A)$ is a multigraph where $V = \{A_1, ..., A_M\}$ is the set of nodes and E_A is the edge set. It can only be defined with respect to a previously defined set of groups, as the one previously described. G_A can be seen as a point in the space of all possible graphs having M vertices and , that we call $\mathcal{G}_A$. Their algebraic representation is a slightly modified version of the adjacency matrix. It is of course a $M \times M$ matrix as that is the size of its set of vertices. The different with respect to a complete graph is that, if the corresponding groups are not a singleton, it can also have 0s along the diagonal. This actually depends on the structure of the groups that are defining the graph.

  The main class for representing those type of graphs is `BlockGraph`, which stores the adjacency matrix as an `RowMajor` **Eigen** matrix, exactly like the `GraphType` class. It is not the only possibility, we also tried different data structure. For example another choice is given by the `BlockGraphAdj` class which stores only the upper triangular part of the adjacency matrix (also called adjacency list) within an **STL** vector. This first option should be preferred as it is more efficient, that second option is however useful for showing how to include a new implementation for a block graph, an argument that is covered at the end of this section .
  The class implementation relies on static polymorphism via *curiously recursive template pattern*

```
template<class D, class T=unsigned int>
class BlockGraphBaseCRTP{
private:
```

```
    /*other methods are available*/
    std::shared_ptr<const Groups> ptr_groups;
};

template<class T=unsigned int>
class BlockGraph : public BlockGraphBaseCRTP<BlockGraph<T>, T>{
public:
    /*other methods are available*/
    inline CompleteView<T> completeview(){
        return CompleteView (*this);
    }
protected:
     Matrix<T,Dynamic,Dynamic,RowMajor> data;
    map<unsigned int, vector<unsigned int> > neighbours;
};

template<class T = unsigned int>
class BlockGraphAdj : public BlockGraphBaseCRTP<BlockGraphAdj<T>, T>{
public:
    /*other methods are available*/
   inline CompleteViewAdj<T> completeview(){
      return CompleteViewAdj (*this);
    }
protected:
    std::vector<T> data;
    map<unsigned int, vector<unsigned int> > neighbours;
};
```

Code 3.9. BlockGraph class

Since a block graph can only be defined with respect to some groups, `BlockGraphBaseCRTP` is not default constructable but has to take as input a shared pointer to a `Groups`, saved as const to be sure that it will not modified. We are using `std::shared_ptr` because a lot of different graphs can be defined with respect to the same groups. This class is also used to define those methods common to all different implementations, the ones that do not directly use values in `data`. We choose to use the *curiously recursive template pattern* in order to increase the generality of the code. In development phase, different data structures had to be tested and hopefully many others would be. Thanks to this kind of implementation, we allow for a reasonably easy implementation of new possible classes without renouncing to an efficient code. Static polymorphism indeed does not introduce the overhead of virtual function calls, which is typical in runtime polymorphism. It could be annoying as those calls are done all over the code, in particular the call operator. Both versions are also move constructable and move assignable.

- The last issue to be discussed is how a block graph is mapped into its complete form.

Indeed, this block form is a useful and synthetic representation of a complete graph. Some operations, such as proposing a new graph, have to be done according to its block form but other computations are defined only with respect to its complete form. In Section 1.1 we described a mapping, called ρ between spaces $\mathscr{G}$ and $\mathscr{G}_A$.

In practice, we decided not to implement such a function. It would indeed be costly to map continuously back and forth big matrices. In other words, there exists no function that maps a `BlockGraph` into a `GraphType`. What we did is instead to equip each block graph with a "View" of its complete form, where a view is simply a class which stores a reference to another class. In this case, `CompleteView` class takes and saves a constant reference to a `BlockGraph` and simply implements a different version of the call operator so that it mimics the ρ operation.

```
template<class T>
class CompleteView{
    using IdxType = typename BlockGraphBaseCRTP<BlockGraph<T>,T>::IdxType;
public:
    CompleteView(BlockGraph<T>const & _G):G(_G), data(_G.data){};

    /*Other methods are available*/

    T operator()(IdxType const & i, IdxType const & j) const
    {
        if(i == j)
            return true;
        else
            return G( G.find_group_idx(i), G.find_group_idx(j) );
    }
private:
    const BlockGraph<T>&  G;
};
```

Code 3.10. CompleteView class

Let us analyse the differences with respect to `operator()` of a block graph. Indices i and j now ranges from 1 to p, the dimension of the complete graph and not from 1 to M as in `BlockGraph`. `find_group_index()` function just accesses the `map_of_indeces` previously described. Finally, what we obtain is a new class called `CompleteView` which does not perform any copy and it implements a call operator such that it is possible to use it as if it were a complete graph even if it is not. Indeed, only elements of the block representation are addressed. There is some overhead due to the need of addressing the elements of `map_of_indeces` vector, but it is still a better solution then performing the actual mapping of the whole matrix.
`CompleteView` does not limit itself to specialize the call operator, its public interface is the same of `GraphType` class. Hence, even if those classes are conceptually different, there is

perfect symmetry between them, which is a key feature of our code. Indeed, the type of graph to be used is always a template parameter, so that every algorithm would be available in two distinct versions, the classical one for complete graphs and the new one for block graphs.
In practice, every type of graph has a method called `completeview()` which returns the graph in its complete view form. For symmetry reasons, this method is available also in `GraphType` but it simply returns itself, since it is already a complete graph. Figure 3.2 summarizes the structure we have just described.

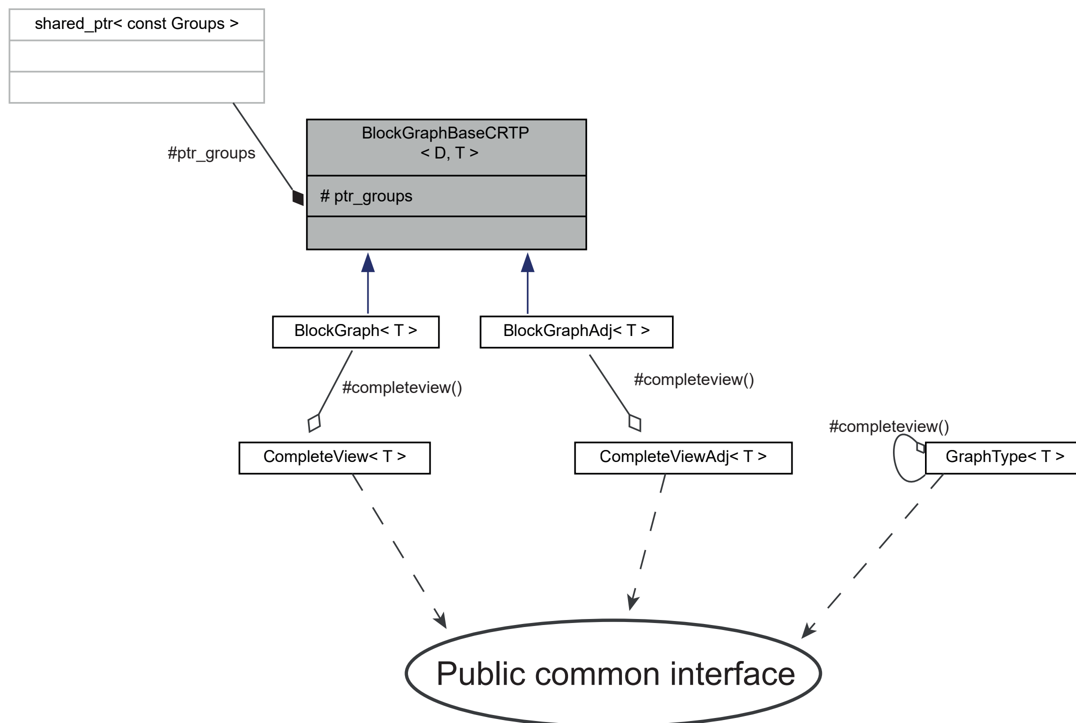

Figure 3.2. Visual representation of graphs and groups hierarchy structure. The block graph classes, called BlockGraph and BlockGraphAdj, are static polymorphic classes deriving from BlockGraphBaseCRTP. They can only be defined with respect to some Groups class, pointed by a shared pointer. The completeview() method map them into their complete forms, called CompleteView and CompleteViewAdj, which can be used exactly as a GraphType class. To ensure perfect symmetry, also GraphType has a completeview() method but in that case it returns the class itself.
This figure is freely inspired by an UML diagram.

What if future developers want to test a new data structure to represent a graph? For a block graph, it is enough to inherit from `BlockGraphBaseCRTP`. If the new type has the same public interface as `BlockGraph`, it can be used everywhere else in the code. This is possible thanks to template programming and by means of our custom type trait `internal_type_traits::isBlockGraph`, whose implementation is done in terms of `std::is_base_of`. In this way, every child of `BlockGraphBaseCRTP` is automatically recog-

nized as a block graph. The only modification one has to do is add the new "CompleteView" type associated to the new block graph to the list of graph types that are identified as complete. This is also needed when adding a new type of complete graph. The way this is very easy to do. For example, if the new type or the new complete view is called NewCompleteType, one just needs to add std::is_same_v<GraphStructure<T>, NewCompleteType<T> > to internal_type_traits::isCompleteGraph implementation.
It is only requested a reasonably simple modification, but we would like to improve it by automatizing even this procedure.

3.2.3 Precision matrix

GWishart class in GWishart.h file is a representation of a $p \times p$ random matrix distributed according to a $\mathrm{GWishart(b, D)}$ distribution, following a "Shape-Inverse Scale" parametrization. We report its density below so that there are no doubts about the parameters meaning.

$$P\left(\boldsymbol{K} \mid G, b, D\right) = I_G\left(b, D\right)^{-1} \left|\boldsymbol{K}\right|^{\frac{b-2}{2}} \exp\left\{-\frac{1}{2} tr\left(\boldsymbol{K} D\right)\right\}, \qquad \boldsymbol{K} \in \mathbb{P}_G \tag{3.3}$$

we have that,

- b is the Shape parameter, it has to be larger than 2 to have a well defined distribution.
- D is the Inverse Scale matrix parameter, it is a $p \times p$ positive definite symmetric matrix.
- $G \in \mathscr{G}$ is a graph with p nodes describing the sparsity structure of the random matrix. It has to be in complete form.
- $\mathbb{P}_G$ is the set of all the $p \times p$ positive definite symmetric matrices constrained by G. $\mathbb{P}_G = \{\boldsymbol{K}, \text{symmetric and positive definite} \mid k_{ij} = 0,\ \forall (i, j) \notin E\}$.
- $I_G(b, D)$ is the normalizing constant, see its definition in (1.7), Chapter 1.

All operations involving such a matrix must respect the constrains imposed by some graph G, which has to be in complete form. Which type of complete graph? Any of the available ones, GraphType and CompleteView should be allowed while a graph in block form like BlockGraph should not. We decided to realize this concept by creating a non-template class such that all its methods involving a graph are actually template methods. This is done to have a more flexible class. If it were a template class, templatized according to the type of constraining graph, once that an instance of GWishart is created, it would always be forced to be used with graphs of that particular type. Instead, a non-template class with template methods allows for using the same method once using one type of complete graph and then another type.

```
class GWishart : public GWishartTraits{
public:
    //Example of one of the available constructors
    template< template <typename> class CompleteStructure = GraphType,
              typename T = unsigned int, typename Derived >
    GWishart(CompleteStructure<T> const & G,
             Eigen::MatrixBase<Derived> const & _data,
             Shape _b, InvScale const & _D)
    {
        static_assert(isCompleteGraph<CompleteStructure,T>::value,
            "___ERROR:_GWISHART_REQUIRES_A_GRAPH_IN_COMPLETE_FORM__");
    }

    //GWishart random matrix sampler
    template<template <typename> class CompleteStructure = GraphType,
             typename T=unsigned int, typename NormType=utils::MeanNorm>
    void rgwish(const CompleteStructure<T> & G, double const threshold,
                sample::GSL_RNG const & engine );

    //Compute the normalizing constant
    template<template <typename> class CompleteStructure = GraphType,
             typename Type = unsigned int>
    long double log_normalizing_constat(const CompleteStructure<Type> & G,
                                        unsigned int const & MCiteration,
                                        sample::GSL_RNG const & engine );
private:
    Shape       b;
    InvScale    D;
    UpperTriCol chol_invD;//chol_invD^T * chol_invD = D^-1
    InnerData   data; //matrix
    UpperTriRow U;    // U = chol(data)^T, i.e, data= U^TU
};
```

Code 3.11. GWishart class

Code 3.11 is a little inside of the class implementation, but it is sufficient for showing the most relevant techniques that were used.

First of all, it shows how graphs are templatized all over the code. Since every graph type class is itself a template class, we decided to fully exploit this feature by imposing that the first template parameter is itself a template. It is always defaulted as `GraphType`. We also refer to a template template parameter, `CompleteStructure` in this example, as a *skeleton* of a graph. Indeed, being a template, it is not possible to create an instance of it, it always has to be completed with some type `T`. In this way, we always keep the skeleton of a graph separated from its type.

How do we ensure that only a complete graph can be passed as template parameter? Of course,

naming the template parameters `CompleteStructure` is of some help but has no effect on the generated code. What we did is to create a customized type trait called `isCompleteGraph` that does exactly what the name says. It returns true if `CompleteStructure` represents a complete graph, false otherwise. What is important is that it does it at compile time. It is then used in combination with `static_assert` to rise a significant error message. It is defined in the `internal_type_traits` namespace.
Code 3.11 also introduces the two most relevant methods of the class. `rgwish()` draws a random sample from distribution (3.3). `log_normalizing_constant()` computes a Monte Carlo approximation of $log(I_G(b, D))$. As previously presented, see Section 2.3, those two function have a central role in GGM sampling strategies and their implementation is exhaustively analysed in the next Section 3.3.
Finally, this class does not just store the random matrix `data` and parameters `b` and `D`, but for efficiency reasons it also computes and stores the Cholesky decomposition of $\texttt{data} = \texttt{U}^T\texttt{U}$ and the Cholesky decomposition of the Scale matrix, i.e the factorization of $\boldsymbol{D}^{-1}$, not of $\boldsymbol{D}$.

3.3 `GWishart` computational aspects

The `utils` namespace collects all free functions used throughout the code. The main ones are `rgwish_core()` and `log_normalizing_constat()`.

3.3.1 A direct sampler for `GWishart` variates

The key ingredient of Algorithm 11 is to be able to draw random samples from GWishart distribution, see (3.3). We rely on the algorithm developed by Lenkoski (2013). It is important to underline that it is a direct sampler, it is not a Markov chain sampler like the block Gibbs Sampler proposed by Piccioni (2000).
Given a graph $G = (V, E)$ having p nodes, let us denote by N_i the neighbourhood of the i-th vertex, see definition (3.2). The goal is to draw a random matrix $\boldsymbol{K} \sim \text{GWishart}(b, D)$, $\boldsymbol{K} \in \mathbb{P}_G$ and it can be done as described in Algorithm 13. It is meant to be a sequential procedure, it can not be accelerated through a parallel implementation. We decided to optimize the way matrix $\boldsymbol{D}$ is inserted as an input and to write a simplest version for the case $|N_i| = 1$. We also put a lot of effort in trying to perform submatrices extraction in the best possible way.

For what concerns the latest point, the difficulty is to select a submatrix defined according a generic index subset of rows and columns N_i, an operation that is usually called *indexing*. This kind of action would become a standard operation as soon as **Eigen 3.4** is released, see indexing and slicing. Nevertheless, since our ultimate purpose is to create a `R` package, it is convenient to rely on the `R` package `RcppEigen`. It provides header files for the library and makes possible automatic conversion from native `R` matrix and the Eigen ones. Unfortunately,

Algorithm 13: A direct sampler for GWishart variates

Step 1. Compute $\boldsymbol{D}^{-1}$ and draw a random matrix $\boldsymbol{K}^0 \sim \text{Wish}(b, \boldsymbol{D}^{-1})$.
Then set $\boldsymbol{\Omega} = \boldsymbol{\Sigma} = (\boldsymbol{K}^0)^{-1}$.

Step 2. For each $i = 1, ..., p$,

2.1 Form $\boldsymbol{\Omega}_{N_i,N_i}$ and $\boldsymbol{\Sigma}_{N_i,i}$. Using a R like notation, they would be defined as $\boldsymbol{\Omega}[N_i, N_i]$ and $\boldsymbol{\Sigma}[N_i, i]$.

2.2 Compute $\boldsymbol{\beta}_i^* \in \mathbb{R}^{|N_i|}$ as

$$\boldsymbol{\beta}_i^* = (\boldsymbol{\Omega}_{N_i,N_i})^{-1}\boldsymbol{\Sigma}_{N_i,i}$$

2.3 Form $\hat{\boldsymbol{\beta}}_i \in \mathbb{R}^{p-1}$ by copying the elements of $\boldsymbol{\beta}_i^*$ to the appropriate location and putting zeros in those location not connected to i in graph G.

2.3 Form $\boldsymbol{\Omega}_{V\setminus i,V\setminus i}$. Using a R like notation it would be $\boldsymbol{\Omega}[-i, -i]$.
Compute $\boldsymbol{\beta}_i = \boldsymbol{\Omega}_{V\setminus i,V\setminus i}\ \hat{\boldsymbol{\beta}}_i$

2.4 Update the i-th row and i-th column of $\boldsymbol{\Omega}$, diagonal element excluded, replacing them with $\boldsymbol{\beta}_i$.
$\boldsymbol{\Omega}_{i,V\setminus i} = \boldsymbol{\beta}_i$, and $\boldsymbol{\Omega}_{V\setminus i,i} = \boldsymbol{\beta}_i^T$, so that $\boldsymbol{\Omega}$ is always symmetric.

Step 3. Repeat step 2 until convergence, then return $\boldsymbol{K} = \boldsymbol{\Omega}^{-1}$.

---

for the moment only version 3.3.7 is supported.

Eigen reference manual explains an advance technique that allows to do the same operation fully exploiting their optimized expressions. It relies on the usage of nullary expresssions.

```
template<class ArgType, class RowIndexType, class ColIndexType>
class indexing_functor {/*Taken from Eigen manual*/};

template <class ArgType, class RowIndexType, class ColIndexType>
Eigen::CwiseNullaryOp<indexing_functor<ArgType,RowIndexType,ColIndexType>,
                      typename indexing_functor<ArgType,
                                                RowIndexType,
                                                ColIndexType>::MatrixType >
indexing(const Eigen::MatrixBase<ArgType>& arg,
         const RowIndexType& row_indices,
         const ColIndexType& col_indices)
{/*Taken from Eigen manual*/};


enum class Symmetric{True,False};
//Wrapping the new expression
template<Symmetric Sym = Symmetric::False>
MatRow SubMatrix(std::vector<unsigned int> const & nbd, MatRow const & M){

    using ArrInt = Eigen::Array<unsigned int, Eigen::Dynamic, 1>;
```

```
    /* Some checks are also done */
    MatRow res=indexing(M,Eigen::Map<const ArrInt> (&(nbd[0]), nbd.size()),
                          Eigen::Map<const ArrInt> (&(nbd[0]), nbd.size()));

    if constexpr(Sym == Symmetric::True){
        return res. template selfadjointView<Eigen::Upper>(); }
    else{/*similarly defined*/}
}
template<class MatType, Symmetric Sym = Symmetric::True>
VecCol View_ExcMult(unsigned int const & x, MatType const & A, VecCol const & b)
{
    static_assert(Sym == Symmetric::True || Sym == Symmetric::False,
    "Error");
    unsigned int p = b.size();
    VecCol res(VecCol::Zero(p));
    if(x == 0){
        if constexpr(Sym == Symmetric::True){
            res = A.bottomRightCorner(p,p)
                   .template selfadjointView<Upper>() * b;}
            else{/*similarly defined*/}
            return res;
        }
    else if(x == p){/*similar to x==0 case*/}
    else{ //general case
        if constexpr(Sym == Symmetric::True){
          res.head(x)=A.topLeftCorner(x,x)
                        .template selfadjointView<Upper>()*b.head(x);
          res.head(x) += A.block(0,x+1,x,p-x) * b.tail(p-x);
          res.tail(p-x)=A.bottomRightCorner(p-x,p-x)
                          .template selfadjointView<Upper>()*b.tail(p-x);
          res.tail(p-x) += A.block(0,x+1,x,p-x).transpose() * b.head(x); }
        else{/*similarly defined*/}
        return res;
    }
}
```

Code 3.12. SubMatrix class

Code 3.12 first presents how the new *nullary expression* is realized and wrapped within a custom class, called `SubMatrix`, that makes the new expression compatible with **STL** vectors. It is used to select matrix $\mathbf{\Omega}_{N_i,N_i}$ and an overloaded version is available for $\mathbf{\Sigma}_{N_i,i}$.
To curry out step 2.3 of Algorithm 13, which consists of this multiplication $\mathbf{\Omega}_{V\setminus i,V\setminus i}\ \hat{\boldsymbol{\beta}}_i$, we use `View_ExcMult()` function. Indeed, as we know in advance which row/column has to be excluded and it would be useless to do $(p-1)*(p-1)$ copies. We actually used values of p such that the benefits of this implementation are not so evident, but it is a competitive alternative to the previous method and it is a nice example on how to use block operations.

The last optimization to be discussed regards the input matrix $\boldsymbol{D}$. As explained in Section 3.2.3 we use a "Shape-Inverse Scale" parametrization for the GWishart. We do not want to modify this choice because it is a standard one in this kind of literature. Unfortunately, computationally speaking, it is not the best choice. `rgwish_core()` would actually prefer a Scale parametrization, which could avoid a matrix inversion or, even better, one would like to insert directly the Cholesky decomposition of the Scale matrix. As explained in Section 3.1.1, the first step of a Wishart random sampler is indeed to factorize the Scale matrix.
From an efficiency point of view, we would prefer the second parametrization but, since this is one of those functions that is exported in `R`, it makes sense to also provide an interface many users are more familiar with. We then decided to allow all those possibilities, taking as input a matrix and defining a template parameter for specifying what kind of matrix it is. An `if constexpr()` clause is used to handle the different scenarios.

```
enum class ScaleForm{Scale,InvScale,CholUpper_InvScale,CholLower_InvScale};
template<template <typename> class GraphStructure = GraphType,
         typename T = unsigned int, ScaleForm form = ScaleForm::InvScale,
         typename NormType = MeanNorm > //Templete parametes
std::tuple< MatRow, bool, int>  //Return type
rgwish_core( GraphStructure<T> const & G, double const & b,
             Eigen::MatrixXd & D, double const & threshold = 1e-8,
             sample::GSL_RNG const & engine = sample::GSL_RNG(),
             unsigned int const & max_iter = 500 )
{
    /*How different types of inputs are handled*/
    MatCol K(MatCol::Zero(N,N));
    if constexpr(form == ScaleForm::Scale){
        K = rwish<MatCol, isChol::False>()(engine, b, D);}
    else if constexpr(form == ScaleForm::InvScale){
        MatCol Inv_D(D.llt().solve(MatCol::Identity(N,N)));
        K = rwish<MatCol,isChol::False>()(engine, b, Inv_D); }
    else if constexpr(form == ScaleForm::CholUpper_InvScale){
        K = rwish<MatCol, isChol::Upper>()(engine, b, D); }
    else if constexpr(form == ScaleForm::CholLower_InvScale){
        K = rwish<MatCol, isChol::Lower>()(engine, b, D); }
    else{throw std::runtime_error("Error");}
};
```

Code 3.13. rgwish_core function

Note that in Code 3.13 we omitted `sample::` before `rwish` and `isChol`. Step 3 of Algorithm 13 states to repeat until convergence, but the literature does not specify with respect to what norm. The last template parameter is used to specify a policy. We created some classes, all implementing a static method called `norm()`. Inside `rgwish_core()` it is enough to call it through `NormType::norm()` and different norms can be called according to what is specified

as template parameter. In any case, the tests did not highlight significant differences.
The remaining part of the implementation follows exactly the procedure of Algorithm 13, a part for the case when $|N_i|= 1$ for which it is possible to further develop some computations analytically which allows to perform less operations. Moreover, this is one of those functions that can have as input parameters only complete graphs. We use the same technique presented in Section 3.2.3 to ensure it, that is a combination of `static_assert` and `internal_type_trait::isCompleteGraph`.

The `rgwish_core()` function does depend on many template parameters. This makes it very flexible without renouncing efficiency. Usually all specified parameters are indeed known at compile time and this is the way it is used all over the code. As mentioned before, we would also like to make it available in `R` but this requires to deduce all parameters at run time. Clearly, this goes against the definition of template functions. The tricky is then to use another function, called `build_rgwish_function()` which returns the version of `rgwish_core()` one would like to use.

```
using rgwishRetType = std::tuple< MatRow, bool, int>;
//Template aliasing
template <template <typename> class GraphStructure, typename T>
using rgwish_function =
      std::function< rgwishRetType(GraphStructure<T> const &,
                        double const &, Eigen::MatrixXd &, double const &,
                        sample::GSL_RNG const &, unsigned int const &) >;

template<template <typename> class GraphStructure = GraphType,
         typename T = unsigned int> //Template parameters
rgwish_function<GraphStructure,T>  //Return type
build_rgwish_function(std::string const & form, std::string const & norm);

/*Usage example*/
auto rgwish_fun = utils::build_rgwish_function<GrapType,int>(form, norm);
auto [Mat, converged, iter] = rgwish_fun(G, b, D, tr, engine, max_iter);
```

Code 3.14. build_rgwish_function

The trick is possible thanks to template aliasing for a `std::function`. At first sight it may result weird but it is very simple to be used. Code 3.14 is an example taken from `BGSL_export.cpp` file, the one that defines the interface with `R`.

We have here presented `rgwish_core()` function. The Suffix *core* is used because this is the most general implementation, both in terms of inputs and outputs. Indeed, it also returns if convergence was reached of not and how many iterations were done. There are some situations in the code where we already know that such information would not be needed, that is the case of `rgwish()` function, which simply calls the previously presented one but only returns the randomly generated matrix.

3.3.2 Computing the normalizing constant

In Section 1.3.1 we discussed some of the problems related to the GWishart normalizing constant. We recall that, in general, it is not available in close analytical form but it has to be approximated. Different methods have been proposed, we now present the computational aspects of our implementation of the Monte Carlo method by Atay-Kayis and Massam (2005).

Although complicated, it is grounded on the idea to exploit the free elements to compute the non-free elements and to approximate I_G as the product of a function of non-free elements and a constant term. See Theorem 1.3.1.2 for further details. The resulting algorithm is presented in Algorithm 14.
We also recall the definition of ν_i^G, first defined in Section 2.3, equation (2.14).

$$\nu_i^G = |\{j : j > i \text{ and } (i,j) \in E\}| = |N_i \cap \{i+1, \dots, p\}| \tag{3.4}$$

For $i = 1, \dots, p-1$. It is equal to the number of 1s in row i, from position $i+1$ up to the end. For convention, $\nu_p^G = 0$.

As pointed out in Atay-Kayis and Massam (2005), ψ_{ij} for $(i,j) \notin E$ in equations (3.5) and (3.6) has to be computed line by line. In that way, given (i,j), all values ψ_{rs}, for $(r,s) \overset{L}{<} (i,j)$, are available for computing $\psi_{i,j}$, where $\overset{L}{<}$ indicates the lexicographic compare. This suggests to store that matrix by rows.

The number of Monte Carlo iterations performed for approximating (3.8) usually ranges from 500 up to 1000, which makes Algorithm 14 computationally very intense. Moreover, it is not very robust in terms of stability, both numerical and for what concerns its Monte Carlo variance. Numerical instabilities rise because very large or very small numbers are involved and compared, in particular when performing operations (3.7) and (3.8).
To face this issue, we wrote a slightly modified version of Algorithm 14 which computes $log(\hat{I}_G^{MC})$ instead of I_G^{MC} and we exploited the log-sum-exp function to reduce the likelihood of overflows and underflows when computing (3.7) and (3.8).

```
double logSumExp(double x, double y){ //Computes log(exp(x) + exp(y))
    (y > x) ? (swap(x,y)) : (/*do nothing*/);
    return x + gsl_sf_log_1plusx(exp(y-x));
}
double logSumExp(std::vector<double> const & v){ // log(sum_{i=1:N}(exp(v_i)) )
    vector<double>::const_iterator it_max = max_element(v.cbegin(), v.cend());
    double res = accumulate(v.cbegin(), v.cend(), 0.0,
                    [&it_max](double const & _res, double const & x)
                    {return _res + exp(x - *it_max);});
    return *it_max + log(res);
}
```

Code 3.15. LogSumExp functions

Algorithm 14: Monte Carlo approximation for $I_G(b, D)$

Step 1. Compute D^{-1} its Cholesky factorization T such that $D^{-1} = T^T T$.

Step 2. Define $h_{ij} = t_{ij}/t_{jj}$ for $1 \leq i \leq j \leq p$.

Step 3. Let N be the number of Monte Carlo iterations, for iter $= 1, ..., N$ repeat the following steps:

Step 3.1 Draw the free elements:

- For diagonal elements, draw $\psi_{ii} \sim \sqrt{\chi^2(b + |\nu_i^G|)}$, for $i = 1, ..., p$.
- For extra-diagonal elements, draw $\psi_{ij} \sim N(0, 1)$, $\forall (i, j) \in E$, $i < j$.

Step 3.2 For $1 \leq i \leq j \leq p$ and $(i, j) \notin E$, derive the remaining elements of ψ in the following way:

- If $i = 0$:

$$\psi_{ij} = -\sum_{k=1}^{j-1} \psi_{ik}\ h_{kj} \tag{3.5}$$

- If $i > 0$:

$$\psi_{ij} = -\sum_{k=1}^{j-1} \psi_{ik}\ h_{kj} - \sum_{r=1}^{i-1} \left(\frac{\psi_{ri} + \sum_{l=r}^{i-1} \psi_{rl}\ h_{li}}{\psi_{ii}} \right) \left(\psi_{rj} + \sum_{l=r}^{j-1} \psi_{rl}\ h_{lj} \right) \tag{3.6}$$

Step 3.3 Compute

$$\mathrm{J(iter)} = \exp\left\{ -\frac{1}{2} \sum_{(i,j)\notin E} \left(\psi_{ij}\right)^2 \right\} \tag{3.7}$$

Step 4. Compute the Monte Carlo mean

$$\hat{\mathrm{J}}_{MC} = \frac{1}{N} \sum_{\mathrm{iter}=1}^{N} \mathrm{J(iter)} \tag{3.8}$$

Step 5. Compute the constant term C_G

$$C_G = \prod_{i=1}^{p} \left((2\pi)^{|\nu_i^G|/2}\ 2^{(b+|\nu_i^G|)/2}\ \Gamma\left(\frac{b + |\nu_i^G|}{2} \right)\ (t_{ii})^{b+|N_i|} \right) \tag{3.9}$$

Step 6. Finally, $I_G(b, D)$ is approximated by

$$\hat{I}_G^{MC} = \ C_G\ \hat{\mathrm{J}}_{MC}$$

We try to achieve an optimized implementation of Algorithm 14 in three ways. First of all, we strive to perform all operations as much cache friendly as possible. Then, a

careful analysis of (3.5) and (3.6) shows that some of those sums are actually repeated and can be efficiently stored. We define $\text{CumSum}(a,b) := \sum_{k=a}^{b-1} \psi_{ak}\ h_{kb}$ for $b > a$ and $a < p$, $b \leq p$. Those sums are temporarily saved in a $(p-1) \times (p-1)$ matrix called Sums such that $\text{Sums}_{a,b} = \text{CumSum}(a, b+1)$ for $a = 1, ..., p-1$ and $b = 1, ..., p-1$, or equivalently, $\text{CumSum}(a,b) = \text{Sums}_{a,b-1}$ for $a = 1, ..., p-1$ and $b = 2, ..., p$.
When matrix T is diagonal, it is possible to develop some analytic calculations which lead to an easier and more efficient code.

Finally, as it is a Monte Carlo integration, it is possible to parallelize the for loop of Monte Carlo iterations. We achieved by means of multi-threading parallelization with **OpenMP**. It was not a trivial task because of the need of using `LogSumExp` function. At each iteration, we compute $\texttt{SS_nonfree_elem(iter)} = \sum_{(i,j)\notin E} \left(\psi_{ij}\right)^2$. A standard implementation of (3.8) would simply require a reduction over `SS_nonfree_elem`. Unfortunately, `LogSumExp` requires to process the whole vector, see the second overloaded function in Code 3.15. This means that each thread has to save its own vector which are then merged when they have all finished their tasks.

```
template<template <typename> class GraphStructure = GraphType,
         typename Type = unsigned int >
long double log_normalizing_constat(GraphStructure<Type> const & G,
                                    double const & b,
                                    Eigen::MatrixXd const & D,
                                    unsigned int const & MCiteration,
                                    sample::GSL_RNG const & engine)
{
    /*Handling parallel implementation*/
    #pragma omp parallel private(vec_ss_nonfree),
                         shared(vec_ss_nonfree_result)
    {
        vec_ss_nonfree.reserve(MCiteration/n_threads);
        for(IdxType iter=0;iter<MCiteration/n_threads;++iter){//MC loop
        /* Fill vector vec_ss_nonfree */
        }//MC loop finished
        #ifdef PARALLELEXEC
        #pragma omp barrier
        #pragma omp critical
        {
            vec_ss_nonfree_result.insert(vec_ss_nonfree_result.end(),
                            std::make_move_iterator(vec_ss_nonfree.begin()),
                            std::make_move_iterator(vec_ss_nonfree.end()) );
        }
        #endif
    }
    //Compute J_MC
    #ifndef PARALLELEXEC
```

```
        std::swap(vec_ss_nonfree_result, vec_ss_nonfree);
    #endif
    result_MC = -std::log(vec_ss_nonfree_result.size()) +
                logSumExp(vec_ss_nonfree_result);
     /*Compute constant term and return*/
}
```

Code 3.16. log_IG parallel implementation

In Code 3.16, `PARALLELEXEC` is a custom pre-processor macro used to ensure that the code is also able to run sequentially. No significant overhead is introduced by swapping with an empty vector.
A criticism about Code 3.16 is for sure that the number of Monte Carlo iterations has to be multiple of the number of available threads. It is not trivial to avoid this issue and, in the end, a couple of iterations less, out of one thousand, is something we can live with.
As for `rgwish_core()`, also this function may accept as input only graphs that are complete. This is check as usual with
`static_assert(isCompleteGraph<GraphStructure,Type>::value,"Stop")`.
Further details can be found in `utils.h`.

3.4 Implemented methods for Structural Learning

Gaussian graphical models (GGM) are models composed of the following ingredients,

- n p-dimensional observations that are modelled as i.i.d and, conditionally on $\boldsymbol{K}$, as normally distributed. Without loss of generality, they are assumed to be zero-mean.
- A prior over the precision matrix $\boldsymbol{K}$, which is assigned conditionally on $\boldsymbol{G}$.
- A prior distribution for the graph $\boldsymbol{G}$, whose role is to impose a sparsity structure on $\boldsymbol{K}$.

For what concerns the choice of the prior on $\boldsymbol{K}$, we are interested in that part of the literature that focuses on using a GWishart prior. We decided to develop a code that does not allow other choices, since changing that prior would lead to models remarkably different, both in terms of statistical properties and in terms of computational needs.
Given this choice, we can expand the Bayesian representation of the model, obtaining

$$\begin{aligned} \boldsymbol{y}_1, \dots, \boldsymbol{y}_n \mid \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\boldsymbol{0},\ \boldsymbol{K}\right) \\ \boldsymbol{K} \mid G &\sim \text{GWishart}\left(b, D\right) \\ G &\sim \pi\left(G\right) \end{aligned} \tag{3.10}$$

Equations (3.10) still represent a very general model. There is indeed freedom in the choice of the graph prior, $\pi(G)$ and in the choice of the search method method, as discussed in

Chapter 2.
BGSL wants to respect such freedoms. Several priors and search algorithms are implemented in such a way that it is reasonably easy to add new options. This is achieved by runtime polymorphism and generic factories for selecting the desired object. We start describing the polymorphic priors.

3.4.1 Graph priors

We decided to distinguish two major groups of graph priors. The first one is called `Complete` and it is the most generic one, as it contains priors which places any possible probability on any graph in $\mathscr{G}$. The second group, `Truncated`, instead is composed of those priors that set a null probability on all those graphs that belong to $\mathscr{G} \backslash \mathscr{B}$, where $\mathscr{B}$ has been introduced in Chapter 1. We already discussed those kind of priors in Section 1.2.
We populated both sets with two priors. `Uniform` and `Bernoulli` belong to `Complete` while `TruncatedUniform` and `TruncatedBernoulli` belong to `Truncated`. Regarding the two Bernoulli priors, we were interested only in the case when all possible links have the same prior probability of inclusion.
All four cases are summarized below. We here stress the difference between G that is a random variable and $g = (V, E)$ that is its realization.

$$\text{Uniform: } \pi_u(G = g) = \frac{1}{|\mathscr{G}|} \quad \forall g \in \mathscr{G} \tag{3.11}$$

$$\text{TruncatedUniform: } \pi_{tu}(G = g) = \begin{cases} \frac{1}{|\mathscr{G}_A|} & \text{if } g \in \mathscr{B} \\ 0 & \text{if } g \in \mathscr{G} \backslash \mathscr{B} \end{cases} \tag{3.12}$$

$$\text{Bernoulli: } \pi_b(G = g) \propto \theta^{|E|} (1 - \theta)^{\binom{p}{2} - |E|} \quad \forall g \in \mathscr{G} \tag{3.13}$$

$$\text{TruncatedBernoulli: } \pi_{tb}(G = g) \propto \begin{cases} \pi_b(\rho^{-1}(G) = \rho^{-1}(g)) & \text{if } g \in \mathscr{B} \\ 0 & \text{if } g \in \mathscr{G} \backslash \mathscr{B} \end{cases} \tag{3.14}$$

Code 3.17 shows an example of the abstract `GraphPrior` class and one of its concrete realizations, in particular it represents prior (3.12).

```
//Base abstract class
template<template <typename> class GraphStructure = GraphType,
          typename T = unsigned int >
class GraphPrior
{
    using Graph = GraphStructure<T>;
public:
    virtual double Prob(Graph const & ) const = 0;
```

```
    virtual double ratio(Graph const & , Graph const & ) const = 0;
    virtual double log_ratio(Graph const & , Graph const & ) const  = 0;
    virtual unique_ptr<GraphPrior<GraphStructure, T>> clone() const = 0;
    virtual ~GraphPrior() = default;
};
//Concrete polymorphic object
template<template <typename> class BlockGraphStructure = BlockGraph,
         typename T = unsigned int>
class TruncatedUniformPrior : public GraphPrior< BlockGraphStructure, T >
{

    using BlockGraphType = BlockGraphStructure<T>;
    static_assert(isBlockGraph<BlockGraphStructure, T>::value , "Erorr");
public:
    double Prob(BlockGraphType const & G) const override{/*...*/}
    double ratio(BlockGraphType const & G_num,
                 BlockGraphType const & G_den) const override{/*...*/}
    double log_ratio(BlockGraphType const & G_num,
                     BlockGraphType const & G_den) const override{/*...*/}
    unique_ptr<GraphPrior<BlockGraphStructure,T>> clone() const override
    {
        return
        make_unique<TruncatedUniformPrior<BlockGraphStructure, T> >(*this);
    }
};
```

Code 3.17. GraphPrior class

As usual, they are templatized according to the type of graph they need to operate on. By definition, it does not make any sense to use a `Truncated` prior on complete graphs. To avoid such cases, which could be difficult to detect as there would not be any compilation error, we use the usual `static_assert` check but using `isBlockGraph` type trait that is the dual of the previously defined `isCompleteGraph` type trait. It also belongs to `internal_type_traits` namespace, even if it was omitted in Code 3.17 along with `std::`.
`GraphPrior` class hierarchy implements the `clone()` method because the (abstract) class `GGM` needs to store a `std::unique_ptr<GraphPrior<GraphStructure, T>>` object and it also needs to be copy constructable and copy assignable.

We also created a generic factory that allows to easily select the desired prior one wants to include in its model. However, looking at definitions (3.11) and (3.13) one can note that there is no need to specify any parameter to define the `Uniform` priors while the `Bernoulli` ones actually need to know the value of θ. This implies that `Bernoulli`and `TruncatedBernoulli` are not default constructable. To create a factory, we then need to resort to the use of *Variadic templates*

```
enum class PriorType{Complete,Truncated};
enum class PriorCategory{Uniform,Bernoulli};

template< PriorType Type, PriorCategory Category,
          template <typename> class GraphStructure = GraphType,
          typename T = unsigned int , typename... Args  > //Template params
std::unique_ptr<GraphPrior<GraphStructure, T>> //return type
Create_GraphPrior(Args&&... args)
{
    static_assert(Type == PriorType::Complete ||
                  Type == PriorType::Truncated,"Error");
    static_assert(Category == PriorCategory::Uniform ||
                  Category == PriorCategory::Bernoulli,"Error");
    if constexpr(Type == PriorType::Complete){
        if constexpr(Category == PriorCategory::Uniform)
            return std::make_unique<UniformPrior<GraphStructure,T> >
             (std::forward<Args>(args)...);
        else
            return std::make_unique<BernoulliPrior<GraphStructure,T>>
             (std::forward<Args>(args)...);
    }
    else{ //Type == PriorType::Truncated
        if constexpr(Category == PriorCategory::Uniform)
           return std::make_unique<TruncatedUniformPrior<GraphStructure,T> >
      (std::forward<Args>(args)...);
        else
           return make_unique<TruncatedBernoulliPrior<GraphStructure, T> >
     (std::forward<Args>(args)...);
    }
}
```

Code 3.18. PriorFactory class

A full implementation of those class is given in `GraphPrior.h`.

3.4.2 MCMC algorithms for GGM

Section 3.4.1 concludes how all components of model (3.10) have been implemented in `BGSL`. From this point on, we start presenting how all those classes collaborate one another in our sampling strategy. First, we present how all Monte Carlo Markov chain methods presented in Section 2.3 have been realised.

The final goal is to develop six different search models; they can be grouped in three different methodologies: **Add-Remove Metropolis Hasting** works with the marginal posterior probability of the graph and therefore need to compute both the prior and the posterior normalizing constants, **Reversible Jumps for GGM** builds a chain on the joint space of

precision and graph, avoiding the computation of the posterior normalizing constant and finally **Double Reversible Jumps for GGM** exploits the exchange algorithm, see Algorithm 9 in Section 2.3.1, for avoiding to deal with both constants. Each of them is available in two versions, one for block graphs and one which makes use of complete graphs.
In practice, this is realized thanks to three template classes, called `AddRemoveMH`, `ReversibleJumpsMH` and `DoubleReversibleJumpsMH`, see Table 3.1. They are templatized according to the type of graph, which also determines which type of search has to be performed. About this point, we want to stress the fact that the version with block graphs and the one with complete graphs are, from a statistical point of view, very different. They have different properties and lead to different findings. It is up to the user to decide which is the most suitable for its application. From a coding prospective, thanks to the usage of template programming, we can use (almost) the same code to obtain both of them.

	AddRemoveMH	ReversibleJumpsMH	DoubleReversibleJumpsMH
Complete	ARMH, see algo 4	RJ, see algo 6	DRJ, see algo 10
Block	Block ARMH, see algo 5	Block RJ, see algo 7	Block DRJ, see algo 11

Table 3.1. Summary of all MCMC methods for GGM. Each column represent a C++ class, on the rows we have the type of input graph.

Before entering the details of the algorithms presented in Section 2.3.3, we need to explain our way of working with different types in uniform way.
The skeleton of the graph, `GraphType`, `BlockGraph` or `BlockGraphAdj`, is, as usual, a template template parameter called `GraphStructure`. It determines which type of algorithm has to be executed, the one for complete graphs or the one for block graphs. Even if the second one is selected, there are some methods, such as the ones described in Section 3.3, that need to take as input graphs in complete form. In Section 3.2.2 we discussed the usage of `completeview()` function. In synthesis, it maps a `BlockGraph` skeleton into a `CompleteView` one, a `BlockGraphAdj` into a `CompleteViewAdj`, and `GraphType` into itself, see Figure 3.2. This mapping is straightforward as long as we know what type of skeleton we are dealing with. We would like to replicate this mapping in abstract terms, that is, given a generic `GraphStructure` we need to define its corresponding `CompleteSkeleton`.
That is exactly what is done in our type trait `internal_type_traits::Complete_skeleton`. It works like as a static `if-else` statement where multiple choices are possible. Indeed, it is implemented in terms of `std::conditional_t` which is a standard type trait implementing an `if-else` condition. We simply perform multiple checks ending up with the desired type. Code 3.19 shows how to use it in practice.

```
template<template <typename> class GraphStructure = GraphType,
         typename T = unsigned int>
struct GGMTraits
```

```
{
template<typename S>
using CompleteSkeleton = typename Complete_skeleton<GraphStructure>::
                                    template CompleteSkeleton<S>;
using CompleteType  = CompleteSkeleton<T>;
};
```

Code 3.19. Usage of Complete_skeleton type trait

Note the use of *template aliasing* to define a skeleton. `template` keyword after the scope operator can not be omitted. `CompleteType` is an example of how this skeleton can be filled with a type to create an concrete type.

Now that also all types are correctly set, we can introduce the implementation details. We realized this symmetric structure by creating a dynamical polymorphic family. Each class implements its specific call operator which performs a single MCMC step, that is made of proposing a new graph, proposing a new precision matrix (this is actually not needed in `AddRemoveMH`), computing the acceptance probability and finally performing the selected move. The abstract base class looks like this,

```
enum class MoveType{Add, Remove};
template<template <typename> class GraphStructure = GraphType,
         typename T = unsigned int>
class GGM : public GGMTraits<GraphStructure, T> {

    public:
        //Some possible constructors
        GGM( std::unique_ptr<GraphPrior<GraphStructure, T>>& _ptr_prior,
             double const & _b , MatCol const & _D, double const & _tr):
                ptr_prior(std::move(_ptr_prior)),  Kprior( _b, _D ),
                trGwishSampler(_tr) {}
        GGM(GGM & _ggm):
                ptr_prior(_ggm.ptr_prior->clone()),
                Kprior(_ggm.Kprior), trGwishSampler(_ggm.trGwishSampler){}
        GGM(GGM &&) = default;

        //Operators
        GGM & operator=(const GGM & _ggm){/*implemented with clone()*/}
        GGM & operator=(GGM &&)=default;

        virtual std::tuple<MatRow, int>
        operator()(MatCol const& data, unsigned int const& n, Graph& Gold,
                   double alpha, sample::GSL_RNG const & engine ) = 0;

        //Propose new graph
        std::tuple<Graph, double, MoveType>
        propose_new_graph(Graph& Gold, double alpha, GSL_RNG const& engine);
```

```
        //Desctructor
        virtual ~GGM() = default;
    protected:
        std::unique_ptr<GraphPrior<GraphStructure, T>>  ptr_prior;
        GWishart Kprior;
        double trGwishSampler;
};
```

Code 3.20. GGM base class

Even though more constructors than the one presented in Code 3.20 are available, `GGM` class is not default constructable. It also stores a `unique_ptr<GraphPrior<GraphStructure, T>>` representing the prior to be used, which implies that the copy constructor and the copy-assign operator have to be overloaded with a new version that exploits the `clone()` method of `GraphPrior`, see Code 3.17. Move semantic operations need to be restored as well.
It is important to remember that in `operator()` the inserted graph is not `const` because it is modified inside the function, if the move is accepted. The new precision matrix and an integer, which stats if the move is accepted or not, are instead explicitly returned.

`AddRemoveMH` class implementation is straightforward as it simply translate into C++ code what is presented in Section 2.1. All details can be found in `AddRemoveMH.h`.
`ReversibleJumpsMH` instead deserves a more accurate description.
For what concerns the way a new graph is proposed, the assembling of the acceptance-rejection probability term and therefore the decision of the move to be performed, it is possible to write a single piece of code for both complete and block graphs. Instead, the way a new precision matrix is proposed and the computation of some terms in the acceptance-rejection probabilities (2.18) and (2.22) strongly depends on the type of the graph and needs to be coded in separate functions. We create an apposite method, called `RJ()`, for constructing the new proposed precision matrix. Its purpose is to perform steps 1.2 and 1.3 for Algorithm 6 if running for complete graphs or step 1.2 and 1.3 for Algorithm 7 if used with block graphs.

It is then called inside `operator()` which completes all other needed steps, which are independent from the type of the graph. In this way, on the front end, it can be used exactly like for `AddRemoveMH`.
We would need a double implementation of `RJ()` but we would also like to maintain a single one for the call operator. The easiest way would be to exploit template specialization, but, since the template parameters are put on the class definition and not on its method, we would need to specialize the whole class. It can be done but it would not be a general solution. Let us suppose, for example, that we specialize the class for the case `GraphStructure` equal to `GraphType`. What if a new complete type were implemented? Would we have to provide a further specialization, even if it would be the same we have for `GraphType`? And what if more

than one type has to be re-defined? We already introduce our type traits `isCompleteGraph` and `isBlockGraph`. The trick here is to use them in combination with the standard type trait `std::enable_if`. It is a complex type trait and an exhaustive explanation of how it works is beyond the purpose of this dissertation. We simply explain how we used it in class `ReversibleJumpsMH`.

```
template<template <typename> class GraphStructure = GraphType,
         typename T = unsigned int >
class ReversibleJumpsMH : public GGM<GraphStructure, T> {
public:
    /*Constructors*/

    //BlockGraph case
    template<template<typename>class GG = GraphStructure, typename TT = T,
             std::enable_if_t<isBlockGraph<GG,TT>::value , TT> =0  >
    std::tuple<PrecisionType, double, double> //Return type
    RJ(CompleteType const & Gnew, PrecisionType& Kold_prior, MoveType Move,
       sample::GSL_RNG const & engine);


    //CompleteGraphs case
    template<template<typename> class GG = GraphStructure, typename TT = T,
             std::enable_if_t<isCompleteGraph<GG,TT>::value , TT> =0  >
    std::tuple<PrecisionType, double, double>
    RJ(CompleteType const & Gnew, PrecisionType& Kold_prior, MoveType Move,
       sample::GSL_RNG const & engine);


    //Call operator
    std::tuple<MatRow, int>
    operator()(MatCol const& data, unsigned int const& n, Graph& Gold,
               double alpha, sample::GSL_RNG const & engine );
protected:
    double const sigma;
    unsigned int MCiterPrior;
};
```

Code 3.21. RJ() method in ReversibleJumpsMH class

`std::enable_if<bool B, class T = void>` itself is actually a simple type trait. It takes two arguments, if the first one is true, it has a public member typedef type, equal to T. `std::enable_if_t` is just a shorter notation to access that typedef. Otherwise, if `B` is false, there is no member typedef and `std::enable_if_t` would then rise an error. Difficulties come because it is usually utilized as a way to leverage SFINAE, *"Substitution Failure Is Not An Error"*. The resulting effect is that, when the member typedef possibly defined by `std::enable_if_t` is used to set a template parameter, it would generate a failure which is

not an error, hence that function is excluded from overload resolution and the compiler would look for other, more suitable, functions.
`RJ()` is defined as a template method whose template parameters are defaulted to be the same of the template class. Their purpose is only to enforce template parameter substitution. It would not work otherwise, SFINAE only kicks in when the substitution for the template parameter fails. The third argument is the `enable_if` condition. It checks if `isBlockGraph` or `isCompleteGraph` are true. If so, a new type is defined but it is defaulted to 0 since it will not be needed. It is admissible since unused template parameters may or may not have a name. It is a sort of "dummy" parameter, necessary only to activate the mechanism of `enable_if`. If both conditions are false, the compiler does not find any suitable function to be called and raises an error like *" error: no type named 'type' in 'struct std::enable_if<false, unsigned int>' "*.
What we achieved in practice is static overload of a method of a template class. Furthermore, the call inside `operator()` is the same as for any other method, Code 3.22 is an example.

```
template< template <typename> class GraphStructure , typename T >
typename ReversibleJumpsMH<GraphStructure, T>::ReturnType
ReversibleJumpsMH<GraphStructure, T>::operator()(
                        MatCol const & data, unsigned int const & n,
                        typename GGMTraits<GraphStructure, T>::Graph & Gold,
                        double alpha, sample::GSL_RNG const & engine)
{

    using Graph  = GraphStructure<T>;
    using CompleteType = typename GGMTraits<GraphStructure, T>::CompleteType;

    //1) Propose new Graph
    auto [Gnew, log_GraphMove_proposal, mv_type] =
                this->propose_new_graph(Gold, alpha, engine) ;
    //2) Perform RJ according to the proposed move and graph
    CompleteType Gnew_complete(Gnew.completeview());
    auto [Knew_prior, log_rj_proposal, log_jacobian_mv] =
                this->RJ(Gnew_complete, this->Kprior, mv_type, engine) ;
     /*3) Compute acceptance probability, perform the move and return*/

};
```

Code 3.22. operator() in ReversibleJumpsMH class

More details are available in `RJMH.h`.

The last and most important MCMC method is implemented in `DoubleReversibleJumpsMH`. As pointed out in Section 2.3, a double reversible jump algorithm **is a** reversible jump such that a further jump has to be done in order to replace the term $I_G(b, D)/I_{G'}(b, D)$. The "Is-a" relationship suggests as to realize it by inheriting from `ReversibleJumpsMH`, see Figure 3.3.

We need to provide an additional member object, the auxiliary matrix $\tilde{\boldsymbol{W}}$, all other machinery needed to perform the move can be inherited. `operator()` is then implemented in terms of the same `RJ()` method we have just discussed, see Code 3.23

```
template< template <typename> class GraphStructure , typename T >
typename DoubleReversibleJumpsMH<GraphStructure, T>::ReturnType
DoubleReversibleJumpsMH<GraphStructure, T>::operator()(
                MatCol const & data, unsigned int const & n,
                typename GGMTraits<GraphStructure, T>::Graph & Gold,
                double alpha, sample::GSL_RNG const & engine )
{
    //1) Propose a new graph
    auto [Gnew, log_GraphMove_proposal, mv_type] =
                this->propose_new_graph(Gold, alpha, engine);
    MoveType inv_mv_type;
    (mv_type == MoveType::Add) ?
        (inv_mv_type=MoveType::Remove):(inv_mv_type=MoveType::Add);

    CompleteType Gnew_complete(Gnew.completeview());
    CompleteType Gold_complete(Gold.completeview());

    //2) Sample auxiliary matrix according to Gnew
    this->Waux.rgwish(Gnew_complete, this->trGwishSampler, engine);
    this->Waux.compute_Chol();
    //3) Perform the first jump, K -> K'
    auto [Knew, log_rj_proposal_K, log_jacobian_mv_K ] =
                this->RJ(Gnew_complete, this->Kprior, mv_type, engine);
    //4) Perform the second jump, Waux -> W0
    auto [W0, log_rj_proposal_W, log_jacobian_mv_W ] =
                this->RJ(Gold_complete, this->Waux, inv_mv_type, engine);

    /*5) Compute acceptance probability ratio, perform the move and return*/
};
```

Code 3.23. operator() in DoubleReversibleJumpsMH class

Steps 3 and 4 of Code 3.23 translate into code the procedure presented in Algorithm 10 and Algorithm 11. Whatever type of jump, dimension increasing or decreasing, is performed at step 3, step 4 has to do the inverse type of move, and this is decided by `mv_type` and `inv_mv_type` variables. A further difference is the graph passed as input. That is the graph whose structure constraints the new proposed matrix. The new precision matrix `Knew` is defined with respect to `Gnew`, i.e the proposed graph, and the new auxiliary matrix $\boldsymbol{W}^0$ with respect to `Gold`, i.e the current graph. See `DRJMH.h` for all coding details.

Wrapping up, we created a polymorphic family whose hierarchy tree is summarized in Figure 3.3. All available search algorithms derive from `GGM.h` class. A single step of each MCMC can be performed by means of the call operator, which has the same firm in all derived

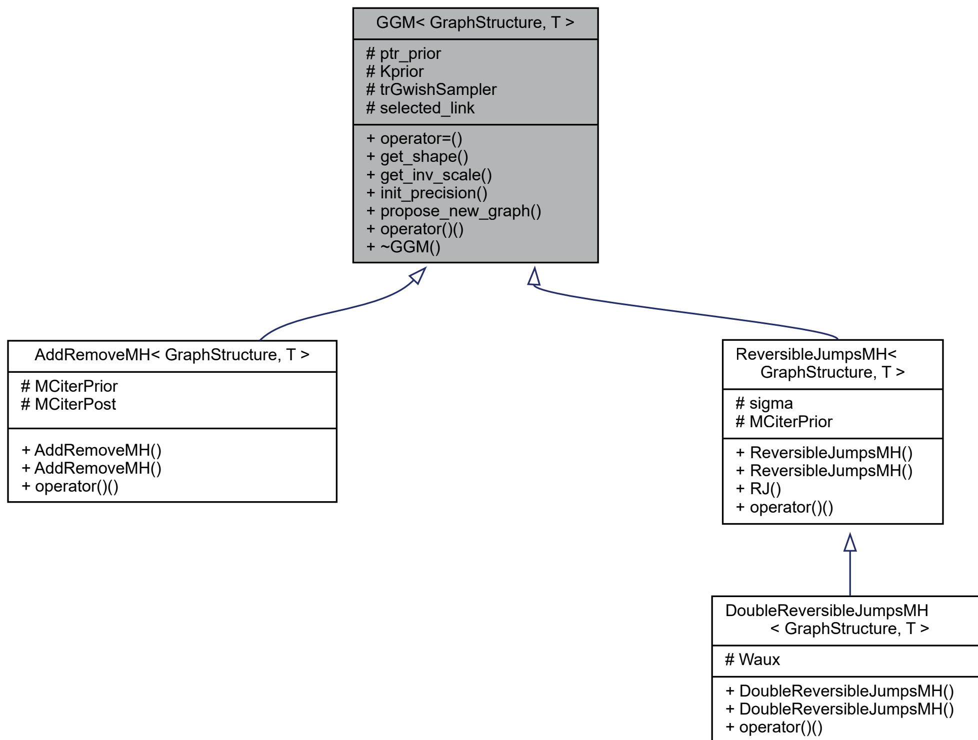

Figure 3.3. `BGSL` hierarchy for Monte Carlo Markov chain implemented methods.

classes, so that it is possible to fully exploit runtime polymorphism. All those methods are templatized with respect to the form of the graph, which defines what kind of version has to run, the classical one with complete graphs or the newly introduced block-wise model.
As we did for graph priors, see Section 3.4.1, we built a factory to facilitate the selection of the method to be used via a `std::unique_ptr< GGM<GraphStructure, T> >`. Since different classes need different constructors, we exploited *variadic templates* the same way we did in Code 3.18.

3.5 BGSL samplers

`BGSL` exposes three samplers. One just for sampling from a GGM, called `GGMsampler`, one for sampling from a Functional Graphical model (FGM for short), called `FGMsampler` and a simpler version of it called `FLMsampler` which is the only one that does not include any graph estimation. It is convenient to frame them in two groups, called *graphical* and *functional* samplers. `GGMsampler` belongs to the first one, `FLMsampler` to the second one and `FGMsampler` is a sort of bridge between them as it is both graphical and functional. This distinction is useful to quickly understand the structure of the output files.
In `BGSL` code organization, they are the most external classes, the ones the users usually want

to use.
As they need many parameters as input, we decided to collect them in structures having more significant names in order to improve readability. Graphical samplers rely on `Hyperparameters`, `Parameters` and `Init` while `FLMsampler` has its own versions called `FLMHyperparameters`, `FLMParameters` and `InitFLM`. They are simply `structs` wrapping useful information for the sampling scheme, they are defined in `SamplerOptions.h` and `FLMSamplerOptions.h` files.
How to use those samplers and a detailed description of the required inputs is given in Section 3.6.6 where we discuss the interface with `R`, as that is actually the way they are meant to be used. Nevertheless, we recall that according to `BGSL` code style guidelines, see the opening of Chapter 3, it would be possible to run the samplers in a pure C++ environment, see for example Code 3.26.

3.5.1 GGMsampler

The purpose of `GGMsampler` is to draw (possibly i.i.d) samples from the posterior distribution of model (3.10). For the sake of clarity, we repeat it below.

$$
\begin{aligned}
\boldsymbol{y}_1, \dots, \boldsymbol{y}_n \mid \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\boldsymbol{0},\ \boldsymbol{K}\right) \\
\boldsymbol{K} \mid G &\sim \text{GWishart}\left(b, D\right) \\
G &\sim \pi\left(G\right)
\end{aligned}
$$

Code 3.24 shows its structure, see file `GGMsampler.h` for a complete implementation.

```
template<template <typename> class GraphStructure = GraphType,
         typename T = unsigned int >
class GGMsampler : public SamplerTraits
{
public:
    using Graph = GraphStructure<T>;
    using CompleteType=typename GGMTraits<GraphStructure,T>::CompleteType;
    using PrecisionType=typename GGMTraits<GraphStructure,T>::PrecisionType;
    using GGMType = std::unique_ptr<GGM<GraphStructure, T>>;

    //Constructor
    GGMsampler( MatCol const & _data, unsigned int const & _n,
                Parameters const & _params,
                Hyperparameters const & _hy_params,
                Init<GraphStructure, T> const & _init,
                GGMType & _GGM_method,
                std::string const & _file_name = "GGMresult",
                unsigned int _seed = 0, bool _print_bp = true);
    int run();
```

```
private:
    void check();
    MatCol data; //p x p
    Parameters params;
    Hyperparameters hy_params;
    GGMType ptr_GGM_method;
    Init<GraphStructure, T> init;
    unsigned int p;
    unsigned int n;
    unsigned int grid_pts;
    sample::GSL_RNG engine;
    int total_accepted{0};
    int visited{0};
    bool print_bp;
    std::string file_name;
};
```

Code 3.24. GGMsampler class

The way it works is straightforward. First of all, it checks that the inserted inputs are coherent with one another using the `check()` method, then it is runned by means of `run()`, which simply calls repeatedly the call operator of the polymorphic `GGM` object, see Section 3.4.

```
GGM<GraphStructure, T> & GGM_method= *ptr_GGM_method; //nicer notation
//Start MCMC loop
for(int iter = 0; iter < niter; iter++){

    int accepted_mv{0};
    //Graphical Step
    std::tie(K, accepted_mv) = GGM_method(data, n, G, p_addrm, engine);
    total_accepted += accepted_mv;
    //Write G and K on file
    if(iter >= nburn && (iter - nburn)%thinG==0 && it_saved < iter_to_store)
    {
        VecCol UpperK{utils::get_upper_part(K)};
        HDF5conversion::AddVector(dataset_Prec, UpperK, it_saved);
        HDF5conversion::AddUintVector(dataset_Graph, adj_file, it_saved);
        it_saved++;
    }
}
```

Code 3.25. GGMsampler run method

The chain starts from the initial values that are provided in `init`. The outputs of `run()` are an integer which counts how many moves were accepted and an HDF file having `".h5"` as extension.

If the execution is stopped by the user, no file is generated and -1 is returned.
Code 3.26 is an example of how it can be used.

```
unsigned int p = 6;
unsigned int n = 500;
unsigned int n_groups = 3;
shared_ptr<const Groups> ptr_groups = make_shared<const Groups>(n_groups,p);
//Simulate data
auto [data, Prec_true, G_true] = utils::SimulateDataGGM_Block(p,n,n_groups);
//Use default parameters
Hyperparameters hy(p); //use default values
Parameters param(); //use default values
Init<BlockGraph> init(n,p, ptr_groups); //start from empty graph
//Select the method to be used
std::string prior = "TruncatedUniform";
std::string algo = "DRJ";
auto method = SelectMethod_Generic<BlockGraph>(prior, algo, hy, param);
//Crete sampler obj
std::string file_name = "GGMexample";
GGMsampler<BlockGraph> Sampler(data, n, param, hy, init, method);
//Run
int accepted = Sampler.run();
//Read results and posterior analysis
auto[Fplinks, FList, Ftraceplot, Fvisited] =
    Summary_Graph(file_name+".h5", param.iter_to_storeG, p, ptr_groups);
```

Code 3.26. Full example of how to run GGMsampler with simulated data

`SelectMethod_Generic` is another factory that simply wraps together the factories presented in Section 3.4.1 and Section 3.4.2 to create a `std::unique_ptr< GGM<GraphStructure, T> >` containing all needed parameters and storing among its member objects a `std::unique_ptr<GraphPrior<GraphStructure, T>>` so that it is ready to be used inside the sampler.
`Summary_Graph` is an utility defined in the `analysis` namespace, which contains all functions needed for the posterior analysis of the sampled values and it is described in Section 3.5.4.

As explained in Section 3.1.3, all samplers generate a binary file where the sampled values are written. In that section we presented the structure of the most general file, the one generated by `FGMsampler`. The one created by `GGMsampler` is composed only of datasets we called /**Info**, /**Precision** and /**Graphs**. The dimensions are the same of the reference case. The name of the sampler saved in /**Sampler** is "GGMsampler" and it automatically stops wrong requests such as computing summaries of the regression coefficients.

3.5.2 `FLMsampler`

`FLMsampler` class, defined in the homonymous file, implements, within a single method, two samplers for two distinct functional models. They both are simplified versions of model (2.50). The same notation is applied. They do not aim to estimate the underlying graphical structure but keep it fixed.
The first model, defined in (3.15), fixes a diagonal graph and sets a Gamma prior on each diagonal element of the precision matrix. This implies that possible interactions between $\boldsymbol{\beta}$ coefficients are not investigated. We will refer to this model by calling it **diagonal**.

$$\begin{aligned}
\boldsymbol{Y}_i \mid \boldsymbol{\beta}_i,\ \tau_\varepsilon^2 &\overset{\text{ind}}{\sim} N_r\left(\boldsymbol{\Phi}\boldsymbol{\beta}_i,\ \tau_\varepsilon^2 \boldsymbol{I}_r\right) \qquad \forall i = 1:n \\
\boldsymbol{\beta}_1,\dots,\boldsymbol{\beta}_n \mid \boldsymbol{\mu},\ \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\boldsymbol{\mu},\ \boldsymbol{K}\right) \qquad \boldsymbol{K} = \text{diag}\left(\tau_1^2,\dots,\tau_p^2\right) \\
\boldsymbol{\mu} &\sim N_p\left(\boldsymbol{0},\ \sigma_\mu^{-2}\boldsymbol{I}_p\right) \\
\tau_j^2 &\overset{\text{ind}}{\sim} \text{Gamma}\left(a_\tau, b_\tau\right) \quad \forall j = 1:p \\
\tau_\varepsilon^2 &\sim \text{Gamma}\left(a_\varepsilon, b_\varepsilon\right)
\end{aligned} \tag{3.15}$$

The second model, defined in (3.16), looks exactly like model (2.50) but keep the graph fixed. Elements of the precision matrix have to be estimated and a GWishart prior is set on the precision matrix. In this framework, the interactions are know *a priori* and do not need to be investigated, but their intensities are. We will refer to this model by calling it **fixed**.

$$\begin{aligned}
\boldsymbol{Y}_i \mid \boldsymbol{\beta}_i,\ \tau_\varepsilon^2 &\overset{\text{ind}}{\sim} N_r\left(\boldsymbol{\Phi}\boldsymbol{\beta}_i, \tau_\varepsilon^2 \boldsymbol{I}_r\right), \qquad \forall i = 1:n \\
\boldsymbol{\beta}_1,\dots,\boldsymbol{\beta}_n \mid \boldsymbol{\mu},\ \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\boldsymbol{\mu}, \boldsymbol{K}\right) \\
\boldsymbol{\mu} &\sim N_p\left(\boldsymbol{0}, \sigma_\mu^{-2}\boldsymbol{I}_p\right) \\
\tau_\varepsilon^2 &\sim \text{Gamma}\left(a_\varepsilon, b_\varepsilon\right) \\
\boldsymbol{K} \mid G &\sim \text{GWishart}\left(b, D\right) \\
&G \text{ fixed}
\end{aligned} \tag{3.16}$$

As priors in models (3.15) and (3.16) are all conjugate or semi-conjugate, the sampling can easily be done through a *Gibbs sampling strategy.* At each step of the Monte Carlo Markov Chain consists of drawing a sample for every variable from each of the full conditionals.
The full conditional distributions for τ_j^2 random variables are,

$$\tau_j^2 \mid \text{all} \sim \text{Gamma}\left(a_\tau\ \frac{n}{2}, \frac{1}{2}\left(2b_\tau + \sum_{i=1}^{n}\left(\beta_{ij} - \mu_j\right)^2\right)\right) \qquad \forall j = 1, \dots, p \tag{3.17}$$

all the others can be found in Section 2.5.

As (3.15) and (3.16) have may terms in common, we decided to create only one class which is templatized according to the choice of the method one wants to run. It interface is given in Code 3.27.

```
enum class GraphForm{Diagonal, Fix};
//Functional Linear Model sampler.
template< GraphForm Graph = GraphForm::Diagonal>
class FLMsampler : public FLMsamplerTraits
{

public:
    FLMsampler( MatCol const & _data, FLMParameters const & _params,
                FLMHyperparameters const & _hy_params,
                InitFLM const & _init,
                std::string const & _file_name = "FLMresult",
                unsigned int _seed = 0, bool _print_pb = true);

    int run();

private:
    void check() const;
    MatCol data; //grid_pts x n
    FLMParameters params;
    FLMHyperparameters hy_params;
    InitFLM init;
    unsigned int p;
    unsigned int n;
    unsigned int grid_pts;
    sample::GSL_RNG engine;
    bool print_pb;
    std::string file_name;
};
```

Code 3.27. FLMsampler class

In this case, no graphical step is performed and therefore there is no need to count the number of accepted moves. `run()` simply returns 0 if the execution ended correctly or -1 if it is stopped by the user. In this case, no file is generated. Its implementation is straightforward, it iteratively constructs parameters defining the full conditionals and then uses functions available in `sample` namespace, see Section 3.1.1, to draw a sample. We make use of `if constexpr()` statement to distinguish parts of the code that are not common between models (3.15) and (3.16) but actually depend on the template parameter.
Since the order variables are extracted is not important in Gibbs sampling, we underline that the most efficient way is to sample first $\boldsymbol{\mu}$. This choice allows to loop over n only once, sampling, in one shot, all $\boldsymbol{\beta}$s and computing $\sum_{i=1}^{n}\left(\boldsymbol{Y}_i - \Phi\boldsymbol{\beta}_i\right)^T\left(\boldsymbol{Y}_i - \Phi\boldsymbol{\beta}_i\right)$ and $\sum_{i=1}^{n}\left(\beta_{ij} - \mu_j\right)^2,\ \forall j = 1:p$,

if needed. Furthermore, this loop can be parallelized. Usage of this class is even simpler than Code 3.26 because no `GGM` object has to be defined.

The structure of the generated file is a simplification of the one presented in Section 3.1.3. Since `FLMsampler` is not a graphical sampler, the /**Graphs** dataset is not included in the file. All the other fields are present, but some care is required for /**Precision**.
Both versions of the `FLMsampler`, diagonal and fixed, save the elements of $\boldsymbol{K}$ in that dataset. For what concerns the second case, there are no differences with respect to what is presented in Section 3.1.3. However, when we sample from model (3.15), only the diagonal elements of the precision matrix have to be saved, i.e τ_j^2, $j = 1, ..., p$. In this case, /**Precision** is a linear dataset of length $p * stored_iter$, just like the one used for $\boldsymbol{\mu}$. Outputs coming from different versions are recognized by the name saved in /**Sampler** which may be "FLMsampler_diagonal" or "FLMsampler_fixed".

3.5.3 FGMsampler

This final sampler combines the previous ones, as it samples from a *functional* model which places a *graphical* description of the structure of the regression coefficients. The corresponding class is called `FGMsampler` and it is implemented in file `FGMsampler.h`. Its purpose is to sample from the posterior distribution of model (2.50), that is also reported below.

$$
\begin{aligned}
\boldsymbol{Y}_i \mid \boldsymbol{\beta}_i,\ \tau_\varepsilon^2 &\overset{\text{ind}}{\sim} N_r\left(\boldsymbol{\Phi}\boldsymbol{\beta}_i, \tau_\varepsilon^2 \boldsymbol{I}_r\right), && \forall i = 1:n \\
\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n \mid \boldsymbol{\mu},\ \boldsymbol{K} &\overset{\text{iid}}{\sim} N_p\left(\boldsymbol{\mu}, \boldsymbol{K}\right) \\
\boldsymbol{\mu} &\sim N_p\left(\boldsymbol{0}, \sigma_\mu^{-2} \boldsymbol{I}_p\right) \\
\tau_\varepsilon^2 &\sim \text{Gamma}\left(a_\varepsilon, b_\varepsilon\right) \\
\boldsymbol{K} \mid G &\sim \text{GWishart}\left(b, D\right) \\
G &\sim \pi\left(G\right).
\end{aligned}
$$

Being a *graphical* sampler, it needs as input a `std::unique_ptr< GGM<GraphStructure, T> >` that defines what MCMC method is to be runned for the graphical step, i.e the sampling of matrix $\boldsymbol{K}$ and graph $\boldsymbol{G}$. Complete graphs version and block graphs version are of course both allowed. As usual, this is chosen at compile time via template programming. The interface is almost the same of `GGMsampler` as they both are *graphical* samplers.

```
template<template <typename> class GraphStructure = GraphType,
         typename T = unsigned int >
class FGMsampler : public SamplerTraits
{
public:
    using Graph = GraphStructure<T>;
```

```
    using CompleteType=typename GGMTraits<GraphStructure,T>::CompleteType;
    using PrecisionType=typename GGMTraits<GraphStructure,T>::PrecisionType;
    using GGMType = std::unique_ptr<GGM<GraphStructure, T>>;

    FGMsampler( MatCol const & _data, Parameters const & _params,
                Hyperparameters const & _hy_params,
                Init<GraphStructure, T> const & _init,
                GGMType & _GGM_method,
                std::string const & _file_name = "FGMresult",
                unsigned int _seed = 0, bool _print_pb = true )

    int run();

private:
    void check() const;
    MatCol data; //grid_pts x n
    Parameters params;
    Hyperparameters hy_params;
    GGMType ptr_GGM_method;
    Init<GraphStructure, T> init;
    unsigned int p;
    unsigned int n;
    unsigned int grid_pts;
    sample::GSL_RNG engine;
    int total_accepted{0};
    int visited{0};
    bool print_pb;
    std::string file_name;
};
```

Code 3.28. FGMsampler class

Our sampling strategy is synthesized in Algorithm 15. As far as Step 1 is concerned, it consists in sampling from the full conditional distributions described in Section 2.5, just like we did for `FLMsampler` in Section 3.5.2. The following one is a *Metropolis-Hasting* graphical step. All possibilities described in Section 2.3 and Section 3.5.1 are available. Although, this time the GGM is not put over the observed data but on the regression coefficients, named $\boldsymbol{\beta}$. In other words, we are using the call operator of the polymorphic family `GGM` as if data were $\boldsymbol{\beta}_1 - \boldsymbol{\mu}, \dots, \boldsymbol{\beta}_n - \boldsymbol{\mu}$. We recall that (3.10) requires zero-mean data. The resulting strategy is called *Metropolis within Gibbs*.

The output of `run()` is composed of the number of accepted moves of Step 2 of Algorithm 15 and the writing of a binary **HDF** file, see Section 3.1.3, containing all sampled quantities and all visited graphs. As for the other samplers, if it is stopped by the users no file is generated and it returns -1.

Code 3.29 contains an example of how to use it. Note that, even if all parameters can be

Algorithm 15: Metropolis within Gibbs sampling strategy for `FGMsampler`

For each iteration:

Step 1. Sample the regression parameters: (2.51).

1.1 Sample the parameter μ from its full conditional given in (2.52)

1.2 Sample the parameters $\boldsymbol{\beta}$'s from their full conditionals given in (2.51).

1.3 Sample the parameter τ_ε^2 from its full conditional in (2.53).

Step 2. Sample $(\boldsymbol{G}, \boldsymbol{K})$ exploiting one of the algorithms presented in Section 2.3.

---

defaulted, matrix $\boldsymbol{\Phi}$ has to be explicitly provided by the user, for example by using function `spline::generate_design_matrix()` described in Section 3.1.2.

```
unsigned int p = 6;
unsigned int n = 500;
unsigned int n_grid_points = 250;
unsigned int n_groups = 3;
shared_ptr<const Groups> ptr_groups = make_shared<const Groups>(n_groups,p);
//Create design matrix
MatCol BaseMat;
std::tie(BaseMat, std::ignore) =
    spline::generate_design_matrix(order,p,0.0,10.0,r);
//Simulate data
auto [data, beta, mu, tau_eps, K, G ] =
    utils::SimulateData_Complete(p,n,r,n_groups,BaseMat);
//Use default parameters
Hyperparameters hy(p); //use default values
Parameters param(); //use default values
Init<BlockGraph> init(n,p, ptr_groups); //start from empty graph
//Select the method to be used
std::string prior = "Uniform";
std::string algo = "DRJ";
auto method = SelectMethod_Generic<BlockGraph>(prior, algo, hy, param);
//Crete sampler obj
std::string file_name = "FGMexample";
FGMsampler<BlockGraph>  Sampler(data, param, hy, init, method,
                                file_name, seed, print_info);
//Run
int accepted = Sampler.run();
//Read results and posterior analysis
MatCol MeanBeta = Matrix_PointwiseEstimate( file_name+".h5",
                                            param.iter_to_store, p, n );
VecCol MeanMu = Vector_PointwiseEstimate( file_name+".h5",
                                          param.iter_to_store, p, "Mu" );
VecCol MeanK_vett = Vector_PointwiseEstimate( file_name+".h5",
```

```
                                                         param.iter_to_storeG,
                                                         0.5*p*(p+1), "Precision" );
double MeanTaueps =  Scalar_PointwiseEstimate( file_name+".h5",
                                                          param.iter_to_store );
auto[Fplinks, FList, Ftraceplot, Fvisited] =
    Summary_Graph(file_name+".h5", param.iter_to_storeG, p, ptr_groups);
```

Code 3.29. Full example of how to run FGMsampler with simulated data

Code 3.29 introduces some functions for posterior analysis, but their definition is delegated to the next section.

3.5.4 Posterior Inference

The purpose of a Monte Carlo Markov chain sampler, as suggested by the name, is to produce chains of values drawn from the posterior distribution of the desired model. The final step is to perform statistical inference by summarizing the chains.
Usually one is interested in two kinds of estimates, pointwise and interval. Let us consider an output from `FGMsampler`, Section 3.5.3, that is the most complete one as it takes into account all functional and graphical quantities. The structure of the generated files is described in Section 3.1.3.
For what concerns $\boldsymbol{\beta}$s, $\boldsymbol{\mu}$, $\boldsymbol{K}$ and τ_ε^2 variables, we take the expected values as pointwise estimates. MCMC theory states that it can be approximated by the sample mean of the sampled values. It also assures that the empirical quantiles converge almost surely to the real quantiles of posterior distributions, which means that interval summaries are computed through quantiles of the desired order. In Bayesian statistics, the range given by the lower and the upper bound is called *credibility interval* or *credibility band*. It is clear that longer chains would generate more accurate estimates, as both rely on empirical means taken with respect to the number of sampled values.

Following the same notation of Section 3.1.3, the functions to perform those summaries are divided in three groups:

```
MatCol
Matrix_PointwiseEstimate(string const & file_name, const int& saved_iter,
                         const unsigned int & p, const unsigned int & n );
std::tuple<MatCol,MatCol>
Matrix_ComputeQuantiles(string const & file_name, int const & stored_iter,
                        unsigned int const & p, unsigned int const & n,
                        double const & alpha_lower = 0.025,
                        double const & alpha_upper = 0.975 );
```

Code 3.30. Summary for beta coefficients

Functions in Code 3.30 both take as input the name of the HDF file (the string has to include the extension ".h5"), the number of basis functions p and the number of curves n, as well as how many iterations were saved. We recall, see Section 3.1.3, that those information are accessible from the /Info dataset of each file.
Given a MCMC output of size N, the first utility of Code 3.30 returns a $p \times n$ matrix such that

$$\hat{\beta}_{ij} = \frac{1}{N}\sum_{t=1}^{N}\beta_{ij}^{(t)} \qquad i = 1, \dots, p \text{ and } j = 1, \dots, n \tag{3.18}$$

where $\hat{\beta}_{ij}$ is the final pointwise estimate for the regression coefficients related to the i-th spline of the j-th curve. $\beta_{ij}^{(t)}$ represents the analogous value sampled at the t-th iteration.
The second function example of Code 3.30 computes two $p \times n$ matrices containing the empirical quantiles of order alpha_lower and alpha_upper for all the $p * n$ $\boldsymbol{\beta}$ coefficients.

```
VecCol
Vector_PointwiseEstimate(string const & file_name, const int& saved_iter,
                         const int & n_elem, string const & vett_type );
std::tuple<VecCol,VecCol>
Vector_ComputeQuantiles(string const & file_name, int const & stored_iter,
                        int const & n_elem, string const & vett_type,
                        double const & alpha_lower = 0.05,
                        double const & alpha_upper = 0.95 );
```

Code 3.31. Summary for mu coefficients and precision matrix K

Functions in Code 3.31 are used both for computing summaries of $\boldsymbol{\mu}$ and precision matrix $\boldsymbol{K}$. Indeed, we recall that only the upper triangular part of the latter is saved, row-by-row diagonal elements included. Code 3.29 shows how to distinguish those cases: set input parameter vett_type to be "Mu" of "Precision" and declare through parameter n_elem the length of the vector to be estimated, that is p in the first case, $\frac{p(p+1)}{2}$ in the second one.
Important remark: the output of the "diagonal" version of FLMsampler produces only p estimates of the precision matrix, that are only the diagonal elements, see Section 3.5.2, model (3.15). In that case, n_elem is p also when computing summaries from "Precision" dataset. See Section 3.5.2 for more details.

Apart from this particular situation, the other samplers derive a pointwise and interval estimate for each diagonal and off-diagonal element of the precision matrix. As already mentioned, since it is symmetric by definition, we only return the upper triangular part of it. How to summarize the values sampled from a GWishart distribution is a common difficulty in the literature. Every draw is defined with respect to one graph. One way to respect such structure is to take, for each entry, the pointwise mean only over those matrices that are all defined with respect to the same graph. This is not a good choice in practice because the space of all graphs is so large that even the most frequent one may be visited a few times.

Instead, what it is usually done to assess estimation of the precision matrix $\boldsymbol{K}$ is to compute the mean values and the credible intervals for each element K_{ij}, based on the MCMC samples (Peterson et al. 2015), leading to a model average estimate (Lenkoski 2013). This approach gives up to possibility of obtaining some elements that, a posteriori are exactly equal to zero but, thanks to the structure imposed by the graph, it is a shrinking estimator and it is therefore able to impose sparsity in the matrix.
A worth mentioning posterior summary technique is presented by Mohammadi and Wit (2015) and implemented in the `R` package `BDgraph`. It still is a model average estimate for $\boldsymbol{K}$, but it exploits the expected holding times $w(\boldsymbol{K})$ as weights in the sum. We recall that they are defined as

$$w\left(\boldsymbol{K}\right) = \frac{1}{\beta\left(\boldsymbol{K}\right) + \delta\left(\boldsymbol{K}\right)} \tag{3.19}$$

where $\beta(\boldsymbol{K})$ is the sum of all birth rates and $\delta(\boldsymbol{K})$ the sum of all the death ones. A proper definition and explanation of those is given in Section 1.3.3. It is an efficient choice because, following the work of Cappé et al. (2003), it exploits the Rao-Blackwell theorem to reduce the variance of the estimator. Summing up, the proposed posterior summary provided by `BDgraph` is

$$\hat{\boldsymbol{K}} = \frac{\sum_{t=1}^{N} w_t(\boldsymbol{K}^{(t)})\boldsymbol{K}^{(t)}}{\sum_{t=1}^{N} w_t(\boldsymbol{K}^{(t)})} \tag{3.20}$$

where $w_t(\boldsymbol{K}^{(t)})$ is the expected holding time for the state characterized by the precision matrix $\boldsymbol{K}$ at the t-th iteration, denoted by $\boldsymbol{K}^{(t)}$.
Once again, this kind of estimator is exclusively applicable to continuous time Markov chains, that is not our case. It is therefore an additional clue of the need of being able to reformulate our Block Double Reversible Jumps approach in a continuous time Birth and Death framework.

In `BGSL`, $\hat{\boldsymbol{K}}$ is obtained by simply averaging over the MCMC samples (Cremaschi et al. 2019; Wang and Li 2012),

$$\hat{\boldsymbol{K}} = \frac{1}{N}\sum_{t=1}^{N} \boldsymbol{K}^{(t)} \tag{3.21}$$

Finally, inference about τ_ε^2 are done similarly, see Code 3.32.

```
double
Scalar_PointwiseEstimate(string const & file_name, const int& saved_iter);
std::tuple<double,double>
Scalar_ComputeQuantiles(string const & file_name, int const & stored_iter,
                        double const & alpha_lower = 0.025,
                        double const & alpha_upper = 0.975  );
```

Code 3.32. Summary for taueps coefficient

Posterior inference of the graph has to be performed with some care. In the ideal case, we would like to approximate its posterior distribution with the relative frequency of visits for each

sampled graph. One way of providing a pointwise estimate of the graph structure is to select the maximum a posteriori strategy, which represents the mode of their posterior distribution. Unfortunately, as noticed in Jones et al. (2005), for problems with even a moderate number of nodes p, the space to be explored is so large that the graph frequency can not be viewed as a solid estimate of its posterior probability because the space of all graphs is so large that each particular graph may be encountered only a few times in the course of the MCMC sampling (Peterson et al. 2015). A more practical and stable solution is instead to estimate the posterior edges inclusion marginally.

As before, let N be the size of a MCMC output, then the posterior inclusion probabilities are estimated as

$$\hat{p}_{ij} = \frac{1}{N} \sum_{t=1}^{N} \mathbb{1}\left((i,j) \in E^{(t)}\right) \tag{3.22}$$

where $\mathbb{1}\left((i,j) \in E^{(t)}\right)$ is the indicator function representing the inclusion of the edge linking nodes i and j in the graph $G^{(t)} = \left(V, E^{(t)}\right)$ visited during the t-th iteration. We call `plinks` the upper triangular matrix having $(\hat{p}_{ij})_{i=1:p,j=i:p}$ as elements, which in practice are simply the proportion of the MCMC iteration after the burnin period in which the edge (i,j) was selected to be part of the graph.

Since it contains the posterior probabilities of inclusion, this matrix represents our uncertainty about including or not a link in the final model.

Pointwise graphical estimate is carried out by selecting all edges whose posterior inclusion probability in (3.22) exceeds a given threshold s. $s = 0.5$ is a possible choice, in analogy with the median probability model of Barbieri and Berger (2004), originally proposed in the linear regression setting. A second possibility is based on the Bayesian False Discovery rate (BFDR; Müller et al. 2007; Peterson et al. 2015)

$$BFDR = \frac{\sum_{i<j}(1-\hat{p}_{ij})\mathbb{1}\left(\hat{p}_{ij} \geq s\right)}{\sum_{i<j}\mathbb{1}\left(\hat{p}_{ij} \geq s\right)}, \tag{3.23}$$

where s is selected so that (3.23) is below 0.05.

Code 3.33 shows the reference function for summarizing the output of visited graphs which are saved on HDF file inside the dataset called `"/Graphs"`.

```
std::tuple<MatRow, map<vector<unsigned int>, int> , VecCol, int >
Summary_Graph(std::string const & file_name, int const & stored_iter,
              unsigned int const & p,
              std::shared_ptr<const Groups> const & groups = nullptr);
```

Code 3.33. Summary for visited graphs

Function `Summary_Graph` in Code 3.33 returns the following objects:

- A plinks matrix, defined in (3.22).
- A std::map which contains all visited graphs along with their absolute frequency of visits. Only the upper triangular part of the graph adjacency matrix is reported. Diagonal elements may or may not be included, it depends on the form of the graph and if the corresponding group is a *singleton* or not. All definitions are in Section 3.2.
- An **Eigen** vector such that the m-th element is equal to the size of graph visited at iteration m.
- An integer which counts how many graphical steps have been accepted during the sampling.

Its last input parameter is a shared pointer to the Groups with respect to block graphs are defined, see Section 3.2. If the sampling is performed by means of complete graphs, set that pointer to null. By doing so, the generated plinks matrix would be $p \times p$ just as what is defined in (3.22). If some groups are actually provided, a $M \times M$ matrix is returned, representing the posterior inclusion probability of each block. M is the number of provided groups.
All details regarding the analysis namespace are contained in PosteriorAnalysis.h file.

The current section concludes the presentation of C++ core code for BGSL.

3.6 The BGSL R package

This section is about how we have interfaced with R all C++ code presented in the first part of the current chapter. We cover all parts of the workflow needed in the creation of a package. How R dependencies are managed, the compilation process, the linkage to external libraries and how C++ code can be integrated inside R environment. Finally, we provide all information for the installation process and a couple of usage example.
As all packages that, in principle, may be submitted to the Comprehensive R Archive Network, or CRAN, which is the public clearing house for R packages, we also need to follow a specific rules and file organization. We recall that all code is publicly available on GitHub repository https://github.com/alessandrocolombi/BGSL.

3.6.1 The DESCRIPTION file

The DESCRIPTION file is the way a package presents itself to the world. It contains the name of the package, who are the authors and their collaborators and how to contact them and who can use it by specifying the license. A brief description of what the package does may also be inserted.

Its most important purpose, however, is to specify the dependencies needed to run it correctly. In particular,

- **Depends:** It is used to require a specific version of R, in our case, R ($\geq$ 4.0.2).
- **Imports:** The already existing packages needed by ours. At this stage, we only need "mathjaxr" which is used to improve the documentation style, allowing us to write complex formulas just like in LaTex environment.
- **Suggests:** As the name says, here is where it is possible to specify some packages whose installation is not mandatory. We suggest packages fields and plot.matrix that provide useful tools for plotting matrices.
- **Linking to:** This field is use to specify those R packages meant to include some C++ headers and libraries needed by BGSL. When possible, this is the best way to proceed as flags for header inclusion and library linkages will be automatically added at compile and linking stage. In our case, we need to specify the following packages,

```
LinkingTo:
    Rcpp (>= 1.0.5),
    RcppEigen (>= 0.3.3.7.0),
    RcppParallel (>= 5.0.2)
```

Code 3.34. LinkingTo field in DESCRIPTION file

The first imported package is used to interface the two programming languages. The second one add header files for **Eigen** library, see Section 3.1. The last one creates a suitable environment for achieving parallelization by including Inter(R) TBB.
We would like to stress that CRAN philosophy is to include only stable versions of those libraries, which sometimes do not coincide with the latest ones, so not all features may be available.
For what concerns **Eigen**, we discussed a little side effect of this choice in Section 3.3.1. A small problem we overcame just writing a couple of extra functions. The drawbacks for **tbb** are more relevant. RcppParallel 5.0.2 includes only version 2016 4.3 which is too old to achieve parallel execution of **STL** algorithms. Intel(R) TBB 2018 would be required, which is something we would like to exploit as soon as possible.
Of course, not all C++ libraries have a corresponding R package that enables this completely automatized procedure. As explained in Section 3.1, BGSL also depends on **GSL** and **HDF5**. Unfortunately, those libraries need to be included and linked explicitly. We postpone how to handle this issue to Section 3.6.4 and Section 3.6.5.

3.6.2 The NAMESPACE file

The NAMESPACE file is an automatically generated text file which defines the public interface of the package.
Usually, developers do not want to allow the user to be able to call every written function, including the auxiliary ones. Being able to expose only the main one helps creating a more efficient and user-friendly interface. In practice, only those functions which definition is preceded by `//' @export` are added to NAMESPACE and therefore accessible to the user.
Just a final remark, `//'@export` syntax is valid only if the documentation is generated by Roxygen2. It is not the only option but it is the one BGSL relies on.

3.6.3 Public interface: the exported functions

As explained in the previous section, exposing all auxiliary functions to the public is not an efficient solution. The users would immediately get lost in such a large amount of possibilities. We only expose the front end of our code, that is composed of the three samplers we introduced in Section 3.5 and those functions needed to correctly set their inputs and to elaborate their outputs.
We rely on Rcpp package for exposing C++ functions to R and to make them available as every other R function. This package simplifies this procedure by creating all correct macros needed to insert the functions in the R workflow. It also defines its own C++ types that are automatically convertible to R type or data structures and to some standard C++ types. For example, `Rcpp::String` is able to read strings from R and it can also be converted into an `std::string`. `Rcpp::List` is a very useful type that allows to use a R-list structure style inside a C++ function. `Rcpp::NumericVector`s are able to read R arrays but unfortunately no conversion is available to map them into `std::vector`s. However, RcppEigen provides an automatic conversion from R vectors and matrices into **Eigen** types.
If all types match correctly, one just needs to add `// [[Rcpp::export]]` before the definition and that C++ function would be callable in R.
Some definitions are needed not to get lost. We divide our functions in three groups:

- **Pure C++ functions:** This set is composed of all those C++ functions that do not use Rcpp. They are completely independent from R. Every method presented in the first part of Chapter 3 belongs to this set.
- **Rccp functions:** We found here all those functions that exploit `//[[Rcpp::export]]` macro which exposes them to the R workflow but do not include them in NAMESPACE file. We collected all of them in the file `BGSL_export.cpp`.
- **Exported functions:** As explained in Section 3.6.2, only functions that use `//'@export`

macro are written in the `NAMESPACE` and therefore available to the users.
This group contains both Rcpp and `R` functions. The latter are all collected in the file `BGSL_export.R`, which can be found in the `R/` subfolder.

All this information are also reported in the documentation. We underline that there could also be *Rccp functions* that are not *exported*. This is useful when we want to write the front end interface in `R` language, but, before calling the *pure C++* version, we also need to do some preprocessing via `Rcpp` to assure that all input arguments are correctly set or mapped into the proper classes.

```
//' A direct sampler for GWishart distributed random variables.
//'
//' A description of the function can be inserted here.
//' @param G description of parameter G.
//' @return description of return object,
//' @export
// [[Rcpp::export]]
Rcpp::List //Return type
rGwish(Matrix<unsigned int, Dynamic, Dynamic, ColMajor> const & G,
       double const & b, MatrixXd & D,Rcpp::String norm = "Mean",
       Rcpp::String form = "InvScale",
       Rcpp::Nullable<Rcpp::List> groups = R_NilValue,
       bool check_structure = false, unsigned int const & max_iter = 500,
       double const & threshold_check = 1e-5,
       double const & threshold_conv = 1e-8, int seed = 0)
{
    sample::GSL_RNG engine( static_cast<unsigned int>(seed) );
    if(!groups.isNotNull()){
        GraphType<unsigned int> Graph(G);
        auto rgwish_fun = utils::build_rgwish_function(form, norm);
        auto [Mat, converged, iter] =
            rgwish_fun( Graph.completeview(), b, D, threshold_conv,
                        engine, max_iter);
        if(check_structure){
            bool check = utils::check_structure(Graph.completeview(), Mat,
                                                threshold_check)  );
            return Rcpp::List::create(Rcpp::Named("Matrix")= Mat,
                                      Rcpp::Named("Converged")=converged,
                                      Rcpp::Named("iterations")=iter,
                                      Rcpp::Named("CheckStructure" = check )
        }
        else{
          return Rcpp::List::create ( Rcpp::Named("Matrix")= Mat,
                                      Rcpp::Named("Converged")=converged,
                                      Rcpp::Named("iterations")=iter );
        }
    else{/*Similarly defined*/}
```

```
}
```

Code 3.35. Example of how to export an Rcpp function with Roxygen2 documentation

Code 3.35 is an exhaustive example of an *Rcpp function* that is also *exported*. Roxygen2 style is straightforward to understand. Moreover, note the usage of `Rcpp::Nullable<Rcpp::List>` to set `groups` null by default. However, the most interesting detail of this example is how to return an `Rcpp::List`. In `R`, lists are objects that can wrap together whatever data structure you want. Here we are returning a `Rcpp::List` composed of one matrix and some integers. At first sight this is not surprising as there exist also in C++ structures that allow to store objects of different types. What is almost unbelievable is that, according to the value of input parameter `check_structure` it is possible to return two or three integers. It looks like we are changing the possible return value of a C++ function! This is a perfect example of how to write C++ code that looks a little but more like `R` code. It is possible only via `Rcpp`.

As a final remark, we recall that, to know what functions are available, the user does not need to search inside `BGSL_export.cpp`, `BGSL_export.R` or `NAMESPACE` files. They are all listed and explained inside the documentation that can be inspected directly from the `R` console using `help()` or `?` commands or consulting the pdf manual of the package.

3.6.4 Compilation

When compiling C++ libraries we usually exploit a set of predefined rules that are contained in a file called `Makefile`. When we want to build an `R` package with source code, things are similar but not identical. `R` itself already provides a templated makefile, called `Makevars`. Although possible, it is definitely not recommended to replace it with a custom file. When we include libraries, as **Eigen**, that are available as `R` packages Section 3.6.1, all proper flags are activated by default. It is the ideal, effortless situation.

As already mentioned, `BGSL` needs to be linked to **GSL** and **HDF5** which requires a further step in compilation process, no default flag is activated but we have to be explicit as usually happens with C++ libraries.

We wrote two extra files, `Makevars` (for Linux) and `Makevars.win` (for Windows). As their differences are minimal, we here only refer to the first one. This local definition of `Makevars` overrides `R` default behaviour, so that it is possible to set non-defaultable flags for external libraries. The way this is done may also be very difficult because those operations are very poorly documented. Fortunately, our needs are minimal and very similar to what we normally do with Makefiles, the result is reported in Code 3.36.

```
#Set compiler standards
CXX_STD = CXX17

#GSL
```

```
GSL_CFLAGS = $(shell gsl-config --cflags)
GSL_LIBS = $(shell gsl-config --libs)

#HDF5
HDF5_CFLAG = $(shell pkg-config --cflags hdf5)
HDF5_LIBS = $(shell pkg-config --libs hdf5)

#Add custom flags
PKG_CXXFLAGS += -DPARALLELEXEC $(GSL_CFLAGS) $(HDF5_CFLAG) -fopenmp
PKG_LIBS +=  $(GSL_LIBS) $(HDF5_LIBS) -lz -fopenmp
```

Code 3.36. Makevars file

3.6.5 Installation

The compilation process described in Section 3.6.4 requires all external libraries to be available, as well as all needed `R` packages. If the latter can be obtained with a single line of code in the `R` console, there is some extra work to do for **GSL** and **HDF5**, especially for Windows users. All installation directives are reported in detail in the `README` file, which is displayed in the front page of `BGSL` repository. We repeat here only the main instructions. We start with how to obtain the two needed external libraries.

Getting `GSL` and `HDF5` - Linux

We here assume that `R` program is already available, as well as `devtools` package. If not, follow the procedure suggested in the `README` file.

Unix-system package managers make the installation process straightforward. Note that all following commands are valid for Ubuntu/Debian users only. On other platforms, it should be easy to use the corresponding package managing tool commands. To download the GNU Scientific Library, it is enough to write on the terminal the following commands,

```
$ sudo apt-get install libgsl-dev
# To test if everything went smoothly, try
$ gsl-config --cflags
    # -I/usr/include #where the header files are
$ gsl-config --libs
    #-L/usr/lib/x86_64-linux-gnu -lgsl -lgslcblas -lm
```

Code 3.37. Commands for installing GSL

HDF5 can be obtained similarly,

```
$ sudo apt-get install libhdf5-dev
# To test if everything went smoothly, try
$ pkg-config --cflags hdf5
```

```
    #-I/usr/include/hdf5/serial
$ pkg-config --libs hdf5
    #-L/usr/lib/x86_64-linux-gnu/hdf5/serial -lhdf5
```

Code 3.38. Commands for installing HDF5

Getting GSL and HDF5 - Window

Windows users will have to leave their comfort zone when installing this package as things get a little more complicated. There are two difficulties here. Installing a package is a nontrivial process which requires a whole series of external tools like make, sed, tar, gzip, a C/C++ compiler etc. collectively known as *toolchain*. All those helpers are immediately available in Linux distribution but they are not on standard Windows boxes and they are to be provided. A second problem is to make **GSL** and **HDF5** available at linking stage when BGSL is compiled. The issue here is that Unix files and directories are organized in a standard way and libraries are usually automatically manged via package managing programs. What has to be done is to exploit the R toolchain provided with Rtools to manage libraries by emulating a Unix procedure. We here assume that R program is already available and Rtools40 is correctly installed. If not, follow the instruction provided in the README file.

Let us start with the GNU Scientific Library. The same procedure is repeated for **HDF5**. The main reference we are following is Rtools Packages, which offers an automatized procedure for both 32 and 64 bits architectures. First of all, one has to open a Rtool Bash terminal, which should be available in C:\rtools40\mingw64.exe, and then clone Rtools Packages directory. From this point on, the procedure is similar to Unix installation, Section 3.6.5. Commands are different but they perform similar operations. Note that the whole process may take a while, up to 30 minutes, because it does not exploit precompiled source code as before, but both libraries are built from source. The needed commands are shown in Code 3.39.

```
$ git clone https://github.com/r-windows/rtools-packages.git
$ cd rtools-packages
#Get GSL
$ cd mingw-w64-gsl
$ makepkg-mingw --syncdeps --noconfirm
$ pacman -U mingw-w64-i686-gsl-2.6-1-any.pkg.tar.xz
$ pacman -U mingw-w64-x86_64-gsl-2.6-1-any.pkg.tar.xz
$ rm -f -r pkg src *.xz *.gz #clean the folder
$ cd ../
#Get HDFR
$ cd rtools-packages
$ cd mingw-w64-hdf5
$ makepkg-mingw --syncdeps --noconfirm
$ pacman -U mingw-w64-i686-hdf5-1.10.5-9002-any.pkg-tar.xz
$ pacman -U mingw-w64-x86_64-hdf5-1.10.5-9002-any.pkg-tar.xz
```

```
$ rm -f -r pkg src *.xz *.gz *.bz2 #clean the folder
$ cd ../
```

Code 3.39. Commands for installation on Windows

Completing BGSL installation

From this point on there are no differences between Linux and Windows. We only need to assure that all R dependencies are available and then we can install BGSL, as in Code 3.40

```
> install.packages(c("Rcpp", "RcppEigen", "RcppParallel", "mathjaxr"))
> devtools::install_github("alessandrocolombi/BGSL")
> library("BGSL")
```

Code 3.40. Final step for installation of R dependencies and BGSL package

3.6.6 How to use BGSL

BGSL is now installed, but there are still some details to be precised to correctly run its samplers. In Section 3.5 we have introduced four possible models we can sample from. They all need some fixed parameters to work properly, as well as the starting points of all chains. Passing explicitly all those values would be extremely tedious, that is why we created some predefined structures to set them, which can be loaded by means of some functions we provide. Default values are usually available. Of course, it would be possible to wrap input parameters into different lists with respect to the one we defined. This approach is however strongly discouraged. Indeed, since many parameters are involved, it is preferred to stick to one uniform notation not to get confused.
It is possible that a sampler may need fewer parameters than the ones included in those lists. In that case, just specify the needed ones, leave the others empty.

Here is a list of information to encompass all those parameters. We clarify their names and meaning, which samplers need them and the functions needed to generate our structures. More details are available in the package documentation. In the following, 1 stands for `FLMsampler` in its diagonal version, 2 for `FLMsampler` in its fixed version, 3 for `GGMsampler` and 4 for `FGMsampler`.

- **sampler_parameters():** Defines a list which contains the following parameters:
  - *MCprior*, the number of iterations for the Monte Carlo approximation of the prior normalizing constant of GWishart distribution, see Section 2.1 and Section 3.3.2. Needed by 3 and 4.

- *MCpost*, the number of iterations for the Monte Carlo approximation of the posterior normalizing constant of GWishart distribution, see Section 2.1 and Section 3.3.2. Needed by 3 and 4.
- *BaseMat*, it represents matrix $\mathbf{\Phi}$ of models (2.50), (3.15) and (3.16), it is defined in Section 3.1.2. It is indeed needed by 1, 2 and 4. This matrix can not be defaulted, but it can be easily generated using `Generate_Basis()`.
- *threshold*, it is the threshold for convergence in GWishart sampler. See Section 3.3.1 for its definition. It is needed by 2, 3 and 4.

• **GM_hyperparameters():** This structure should not be used for those samplers that are not *graphical*, i.e not for 1 and 2. Prefer to use the next function for those.
It defines a list which contains the following parameters:

- *a_tau_eps:* Shape parameter for Gamma prior distribution of τ_ε^2 parameter in model (2.50). Needed by 4.
- *b_tau_eps:* Rate parameter for Gamma prior distribution of τ_ε^2 parameter in model (2.50). Needed by and 4.
- *sigma_mu:* Covariance for normal multivariate distribution of $\boldsymbol{\mu}$ parameter in model (2.50). Needed by 4.
- *b_K:* Shape parameter for GWishart prior distribution of $\boldsymbol{K}$ matrix in models (2.50), (3.10). Needed by 3 and 4.
- *D_K:* Inverse-Scale matrix for GWishart prior distribution of $\boldsymbol{K}$ matrix in models (2.50), (3.10). Needed by 3 and 4.
- *p_addrm:* Probability of proposing a new graph by adding a link. See equation (2.2) Section 2.1 for its definition. Needed by 3 and 4.
- *sigmaG:* Standard deviation used to perturb the elements of the precision matrix when constructing the new proposed matrix. See Algorithm 6, Algorithm 7, Algorithm 11 in Section 2.3 . Needed by 3 and 4.
- *Gprior:* Prior probability of inclusion of each link in Bernoulli graphical priors. It is called θ in (3.13) and (3.14). See its definition in Section 3.4.1. Needed by 3 and 4.

• **LM_hyperparameters():** This structure should not be used for *graphical* samplers, i.e not for 3 and 4. Prefer to use the previous function for those.
It defines a list which contains the following parameters:

- *a_tau_eps:* Shape parameter for Gamma prior distribution of τ_ε^2 parameter in models (3.15) and (3.16). Needed by 1,2.

- *b_tau_eps:* Rate parameter for Gamma prior distribution of τ_ε^2 parameter in models (3.15) and (3.16). Needed by 1,2.
- *sigma_mu:* Covariance for normal multivariate distribution of $\boldsymbol{\mu}$ parameter in models (3.15) and (3.16). Needed by 1,2.
- *b_K:* Shape parameter for GWishart prior distribution of $\boldsymbol{K}$ matrix in models (3.15) and (3.16). Needed by 2.
- *D_K:* Inverse-Scale matrix for GWishart prior distribution of $\boldsymbol{K}$ matrix in models (3.15) and (3.16). Needed by 2.
- *a_tauK:* Shape parameter for Gamma prior distribution of τ_j^2 parameters in models (3.15). Needed by 1.
- *b_tauK:* Rate parameter for Gamma prior distribution of τ_j^2 parameters in models (3.15). Needed by 1.

- **GM_init():** This function is used to define the initial values of the chains of the *graphical* samplers, i.e 3 and 4. It should not be used for 1 and 2, prefer the next function in that case.
  By default, all chains start from null values but it is also possible to set random starting points.
  - *G0:* Initial graph when sampling from models (3.10) and (2.50). Needed by 3 and 4.
  - *K0:* Initial precision matrix when sampling from models (3.10) and (2.50). Needed by 3 and 4.
  - *Beta0:* Initial values of all $\boldsymbol{\beta}$ coefficients when sampling from model (2.50). Needed by 4.
  - *mu0:* Initial values of $\boldsymbol{\mu}$ coefficient when sampling from model (2.50). Needed by 4.
  - *tau_eps0:* Initial value of τ_ε^2 precision variable when sampling from model (2.50). It has to be strictly positive. Needed by 4.

- **LM_init():** This function is used to define the initial values of the chains of samplers 1 and 2. It should not be used for 3 and 4, prefer the previous one in that case.
  By default, all chains start from null values but it is also possible to set random starting points.
  - *K0:* Initial precision matrix when sampling from model (3.16). Needed by 2.
  - *Beta0:* Initial values of all $\boldsymbol{\beta}$ coefficients when sampling from models (3.15) and (3.16). Needed by 1 and 2.

- *mu0:* Initial values of $\boldsymbol{\mu}$ coefficient when sampling from models (3.15) and (3.16). Needed by 1 and 2.
- *tauK0:* Initial values for $\tau_1, \dots, \tau_p$ precision variables when sampling from model (3.15). Needed by 1.
- *tau_eps0:* Initial values of τ_ε^2 precision variable when sampling from models (3.15) and (3.16). It has to be strictly positive. Needed by 1 and 2.

We can finally conclude by providing an example of how BGSL can be used.

```
library(BGSL)
#Declare dimension
p =  30 #number of basis
n_groups =  15 #number of groups
n = 500 #number of curves
r = 250 #number of grid points
range_x = c(100,200) #domain for the curves

# Create random graph and precision matrix
form   = "Block"
Glist  = Create_RandomGraph(p = p, n_groups = n_groups, form = form,
                            groups = CreateGroups(p,n_groups) )
# Simulate curves
sim = simulate_curves(p = p, n = n, r = r, range_x = range_x,
                      G = Glist$G_Complete, K = NULL, tau_eps = 0,
                      D = 0.1*diag(p), n_plot = n, n_picks = 2 )

#Get data and design matrix
data = t(sim$data)
BaseMat = sim$basemat

# Set parameters and random initial values
hy = GM_hyperparameters(p = p, sigmaG =  0.5) #default the others
param = sampler_parameters(threshold = 1e-14, BaseMat = BaseMat)
init = GM_init(p = p, n = n, empty = F, form = form, n_groups = n_groups)

# Select GGM method
algo    = "DRJ"
prior   = "Uniform"
form    = "Block"

#Run
niter     =   600000
nburn     =   200000
thin      =   20
thinG     =    1
file_name = "FGMtest"
```

```
result = FGM_sampling(p = p, data = data, niter = niter, burnin = nburn,
                      thin = thin, thinG = thinG,
                      Param = param, HyParam = hy, Init = init,
                      file_name = file_name,
                      form = form, prior = prior, algo = algo,
                      n_groups = n_groups, print_info = T)
```

Code 3.41. How to use FGMsampler

In the `/inst` subfolder we provide some `R` example scripts for running the different samplers and other `BGSL` utilities.

Chapter 4

Numerical Experiments

In this chapter we present some experiments to test `BGSL`. We start validating the correct implementation of the most important utilities, that are the GWishart sampler and the Monte Carlo method to approximate its normalizing constant.

Then we present some simulation studies to test all three provided samplers, `GGMsampler`, `FLMsampler` and `FGMsampler`. Those experiments are not only meant to assess their correct implementation but we also investigate their robustness and their peculiarities with respect to competitors. Therefore, they provide an exhaustive overview of the models capabilities, they assess the level of reliability of the estimates as well as some weak spots and those aspects that still have room for improvement.

Finally, we conclude providing a case study to show the `BGSL` potential in facing up problems from an innovative prospective.

We take the `R` package `BDgraph`, (Mohammadi and Wit 2019) , as a benchmark for evaluating our performances. Even though its first release on the CRAN is very recent, January 2019, it has rapidly established itself as one of the reference tools for handling Gaussian graphical models. It makes available an exact sampler from GWishart distribution and a function for approximating its normalizing constant, just like the ones we presented in Section 3.3. Actually, our implementation is inspired by the one of `BDgraph`.
The way it samples from a GGM model, see (3.10), is instead completely different from our approach. It implements a Monte Carlo Markov chain method, which is not an acceptance-rejection procedure, but it exploits a *Birth and Death* process like the one we discussed in Section 1.3.3. However, since the final goal is the same, some comparisons can be done.

4.1 Validation of `GWishart` utilities

We start validating our implementation of the GWishart sampler and the way we approximate the normalizing constant I_G, both algorithms are described in Section 3.3.

To validate the sampler, we proceed as Lenkoski (2013) when the algorithm has first been proposed.
We set $p = 4$ and G as a four cycle graph where edges $(1, 4)$ and $(2, 3)$ are missing, see Figure 4.1. Given this graph, we then consider sampling $\boldsymbol{K} \sim \text{GWishart}(b, \boldsymbol{D})$ where $b = 103$ and

$$\boldsymbol{D} = \begin{bmatrix} 136.431 & -10.15 & 8.027 & 2.508 \\ -10.15 & 93.417 & -2.122 & -16.162 \\ 8.027 & -2.122 & 116.652 & 11.62 \\ 2.508 & -16.162 & 11.62 & 120.203 \end{bmatrix}$$

which was generated to resemble the posterior distribution after observing 100 draws from $N_4(\boldsymbol{0}, \boldsymbol{I}_4)$. To validate the algorithm, Lenkoski (2013) compares the expectation of $\boldsymbol{K}$, taken over a large number of repetitions, with the output obtained from the *block Gibbs sampler* proposed by Piccioni (2000). This is a valid sampler for the GWishart distribution but it is not a direct one. It was run for 10 million iterations with an additional one million iterations as burn-in. Such a large number of iterations should be expected to characterize $\text{GWishart}(b, \boldsymbol{D})$. The generated output was

$$\begin{bmatrix} 0.7788 & 0.0827 & -0.0516 & 0 \\ 0.0827 & 1.1594 & 0 & 0.1528 \\ -0.0516 & 0 & 0.9122 & -0.0864 \\ 0 & 0.1528 & -0.0864 & 0.9025 \end{bmatrix}$$

We follow the same procedure for validating our implementation. The sample mean taken over 10 million iterations produced the following output,

$$\begin{bmatrix} 0.7788 & 0.0827 & -0.0516 & 0 \\ 0.0827 & 1.1594 & 0 & 0.1528 \\ -0.0516 & 0 & 0.9122 & -0.0863 \\ 0 & 0.1528 & -0.0863 & 0.9024 \end{bmatrix}$$

that is reasonably similar to the previous one.

We also compared the execution time of our function with the one given in `BDgraph` package. Comparison is carried out using `rbenchmark` package (Kusnierczyk 2012), the output is reported in Table 4.1.

As both packages exploit the same algorithm, the difference can only be in terms of their implementation. `BDgraph` is mainly written in C, exploiting **LAPACK** library for linear algebra manipulations, but it is clear that it is totally outperformed by `BGSL`.

Validation of the implementation of the Monte Carlo method by Atay-Kayis and Massam (2005) for the approximation of the normalizing constant is done similarly.

test	replications	elapsed	relative	user.self	sys.self
BDgraph	1000	0.396	14.143	1.558	0.008
BGSL	1000	0.028	1.000	0.028	0.000

Table 4.1. Benchmark comparison between execution time of `BDgraph` and `BGSL` exact sampler for `GWishart` distribution

We fix $p = 6$ and the underlying graph is displayed in Figure 4.1. Shape and Inverse-Scale parameters are chosen as $b = 3$ and

$$D = \begin{bmatrix} 78.084 & -8.831 & -39.253 & -31.959 & -0.429 & -8.289 \\ -8.831 & 6.597 & 4.514 & 3.514 & 0.721 & 1.150 \\ -39.253 & 4.514 & 36.598 & 8.963 & 13.194 & -5.765 \\ -31.959 & 3.514 & 8.963 & 41.763 & -20.965 & 25.707 \\ -0.429 & 0.721 & 13.194 & -20.965 & 47.365 & -22.246 \\ -8.289 & 1.150 & -5.765 & 25.707 & -22.246 & 41.247 \end{bmatrix}$$

which, as before, was generated to resemble the posterior distribution after observing 100 draws from $N_6(\mathbf{0}, \boldsymbol{I}_6)$. We are computing the posterior normalizing constant, see Section 2.1. We chose to show this example rather than a prior constant because the algorithm is less stable in this case and it is possible to observe how the mean over 100 repetitions stabilizes only if enough iterations are performed.

Since graph $\boldsymbol{G}$ can be also seen as a complete form of a block graph, i.e it belongs to $\mathscr{B}$, we

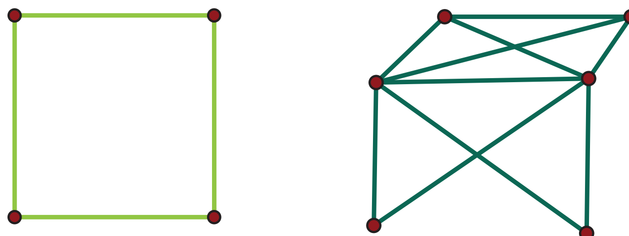

Figure 4.1. The graphs used to validate `BGSL GWishart` direct sampler and the Monte Carlo method for computing its normalizing constant. On the left, we have a four-node cyclic graph, on the right a six node block graph.

can also test the block graph version of our function. The empirical means have to be the similar to the ones of the complete form version, and it is indeed what we see in Figure 4.2. Finally, Table 4.2 reports a benchmark comparison with the analogous function from `BDgraph`. For that comparison, we fixed the number of Monte Carlo iterations to 1000. Even in this case, `BGSL` is faster even though not as much as in the previous example. Indeed, we put much more effort in optimizing `rgwish_core()` function with respect to `log_normalizing_constant()`, see Section 3.3, because, as explained in Section 2.3 our proposed model is able to avoid any normalizing constants computation, see Algorithm 11.

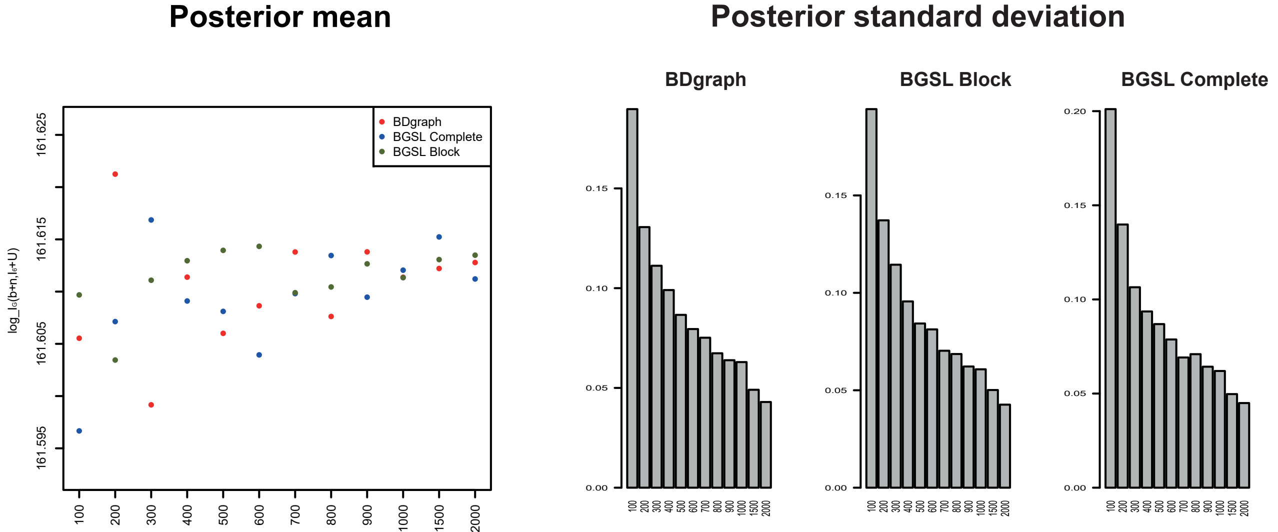

Figure 4.2. Monte Carlo empirical mean for posterior normalizing constant (left) and the corresponding standard deviation (right). The underlying graph has $p = 6$ nodes. Plots are given in logarithmic scale, obtain with 500 repetitions for each possible choice of Monte carlo iterations.

test	replications	elapsed	relative	user.self	sys.self
BDgraph	1000	0.938	1.709	0.924	0.012
BGSL	1000	0.549	1.000	2.150	0.012

Table 4.2. Benchmark comparison between execution time of BDgraph and BGSL MC approximation of posterior normalizing constant

4.2 Testing GGMsampler

In this section we present some tests that aim to validate our `GGMsampler`, in particular when running our proposed Monte Carlo Markov chain method, called **Block Double Reversible Jumps**, or Block DRJ for short. We exploit the `R` package `BDgraph` to perform a challenging comparison, it implements a sophisticated *Birth and Death* MCMC chain, see Section 1.3.3, which has a significantly different theoretical background with respect to our approach. For the moment, we limit ourselves to simulation studies, a case study is given in Section 4.5.

If not explicitly said otherwise, we used a uniform non informative prior for the graph and for the precision matrix we set GWishart$(3, \boldsymbol{I}_p)$, where p depends on the situation.
All final estimates, both from `BGSL` and `BDgraph` outputs, were obtained by cutting the posterior probability of inclusion of each link with the threshold chosen via *Bayesian False Discovery Rate*, presented in Section 3.5.4.
Finally, to assess the performance of the estimated graph structure, we compute the standardized Structural Hamming Distance (Standardized-SHD, Tsamardinos et al. 2006) from the real underlying graph, which in the case of undirected graphs is simply equal to the number of

wrongly estimated links, standardized with respect to the number of all possible ones

$$\text{Standardized-SHD} = \frac{\text{FP} + \text{FN}}{\binom{p}{2}} \tag{4.1}$$

and, inspired by Mohammadi and Wit (2015) and Kumar et al. (2020), we also take in consideration the F_1-score (Baldi et al. 2000; Powers 2020) which is defined as

$$\text{F}_1\text{-score} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}} \tag{4.2}$$

where in (4.1) and in (4.2), TP, TN, FP, FN are the number of true positive, true negative, false positive and false negative, respectively.
Both indices lie between 0 and 1: for the Standardized-SHD lower values are better and indeed 0 stands for perfect estimation, while for F_1-score higher values represent better performances, 1 is for perfect identification.
The difference between them is that (4.1) equally weights errors due to false positiveness or negativeness while (4.2) places higher importance on the number of discoveries made, that are the true positive. To visualize their difference, consider the following simple example: set the true graph to have a sparsity index equal to 0.1 and consider the trivial estimator, i.e the empty graph. Its Standardized-SHD score would be equal to 0.1 that seems reasonably low. On the other hand, the F_1-score score is not deceived as it is equal to 0, recognizing it is a bad estimate.

Following the same approach of Wang (2010) and Mohammadi and Wit (2015), to assess the performance of the precision matrix estimation, we use the Kullback-Leibler divergence (Kullback and Leibler 1951), KL for short, which is defined as

$$\text{KL}\left(\boldsymbol{K}_{\text{true}}, \hat{\boldsymbol{K}}\right) = \frac{1}{2}\left[\langle\, \boldsymbol{K}_{\text{true}}^{-1},\ \hat{\boldsymbol{K}}\, \rangle - p - log\left(\frac{|\hat{\boldsymbol{K}}|}{|\boldsymbol{K}_{\text{true}}|} \right) \right] \tag{4.3}$$

where $\langle\,\cdot\,,\,\cdot\,\rangle$ denotes the trace of the product of two symmetric matrices, $|\cdot| = det(\,\cdot\,)$, $\boldsymbol{K}_{\text{true}}$ is the true underlying matrix used to generate the data and $\hat{\boldsymbol{K}}$ is its estimate. The KL is positive valued such that the minimum is 0 and smaller is better.

- **Experiment 1 - 40 nodes and 20 groups**
  We test `GGMsampler` in a high dimensional case, setting $p = 40$, therefore the number of possible links is 780 which implies $|\mathscr{G}| = 2^{780}$. Data are obtained form $n = 500$ observations from model (1.9). The true underlying graph is itself a block graph, see Figure 4.3. The number of groups is $M = p/2$ that leads to off-diagonal blocks of size $2x2$.
  Since we are dealing with an high dimensional case, we use a Double Reversible Jump method which, as explained in Section 2.3.3, does not need to compute any GWishart

normalizing constant. We test it in both available versions, using graphs in complete form (DRJ) and using graphs in block form (Block DRJ).

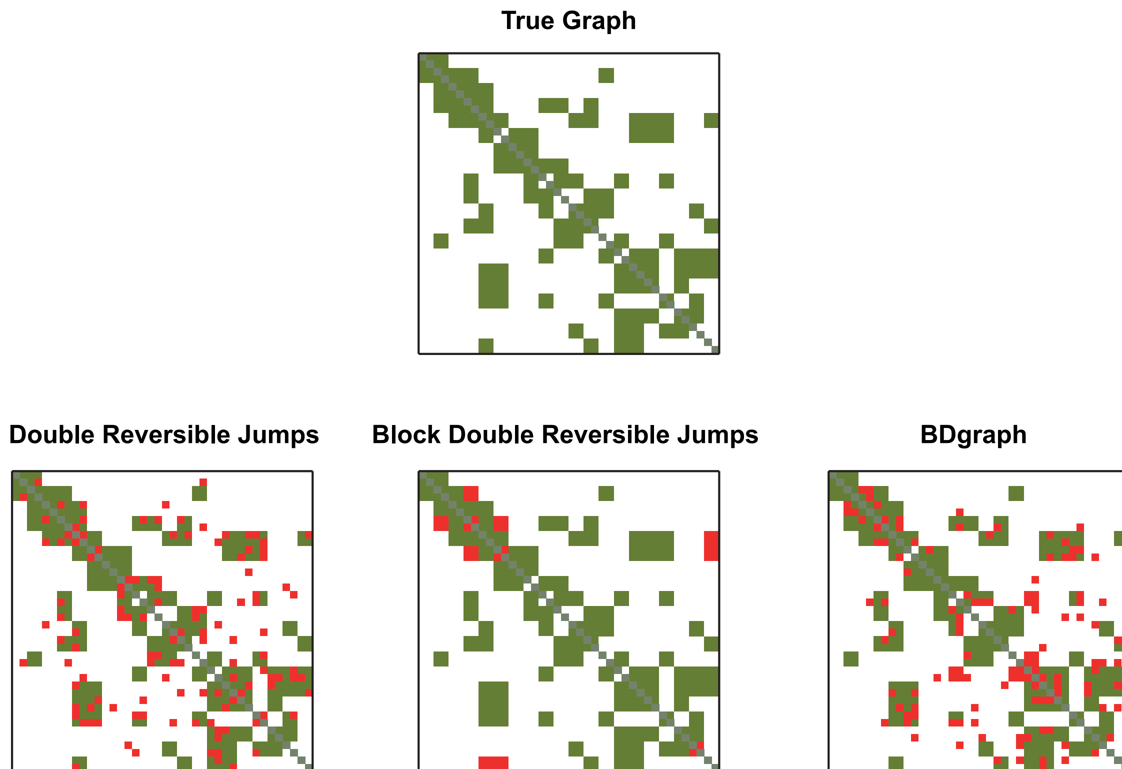

Figure 4.3. The true underlying graph (upper panel) and the estimated graphs. The left-hand panel and the middle one were obtained using BGSL package, the right-hand one using BDgraph. Green squares represent the included links, the red squares stand for edges that are wrongly estimated.

Figure 4.3 compares the final estimates of the forecited methods and the one obtained via `BDgraph`. The upper panel represents the true graph, the green squares mean that there is a link between the corresponding nodes. The middle panel is the proposed Block DRJ method. A simple visual inspection suggests that it is more precise than the two competitors. The number of misclassified edges is rather low, Stamdardized-SHD = 0.0243, and it is well balanced between false positiveness (10) and negativeness (9). Many true discoveries are achieved and indeed its F_1-score = $\frac{400}{419}$ = 0.954 is very high. For what concerns the complete search methods, DRJ (left panel) and `BDgraph` (right panel), they arrive to graphs that are very similar but worse with respect to both indices; the Stamdardized-SHD are equal to 0.0784 and 0.0784 while the F_1-score are 0.84 and 0.836, respectively. The number of links wrongly estimated is the same for both methods (62) but the DRJ makes 4 discoveries more than `BDgraph`, which implies a slightly higher F_1-score. The difference is however not very significant and it is a good validation for the `BGSL` implementation of such method.

The final estimate for the Block DRJ is, according to both indices, closer to the real underlying graph than the two competitors. The reason is that it performs a search in a smaller space, $|\mathscr{B}|= 2^{110}$, hence it manages to find higher probability regions.
The others two do not recognize the block structure of the true graph, they do not even look for such a structure but they try to estimate every possible link independently from the others. This entails more errors in the final estimate as well as a less informative structure of the graph. It would be hard to explain why there are missing edges within some structures that are clearly blocked ones.

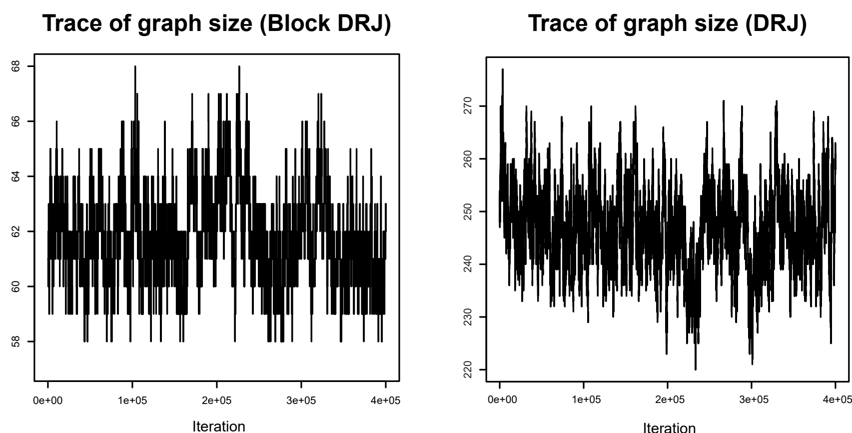

Figure 4.4. Traceplot of the size of the visited graphs, they are obtained via BGSL - Double Reversible Jump method for block graph (left) and complete graphs (right).

All chains were composed of 400′000 iterations plus 100′000 extra iteration as burn-in period that were discarded.
Figure 4.4 shows the traceplot of graph sizes for Block DRJ (left panel) and DRJ (right panel). Both chains reached convergence, but in the first one we can see that it suffers from a low acceptance rate while the one for complete graphs is as thick as a traceplot should be. In this second case we probably could have done a slightly longer burn-in phase.

To get a summary of the precision matrix, we took the empirical mean and empirical quantiles for the values sampled by the MCMC procedure. Note that in this way we do not ensure the final estimate to be block structured but we can not use only those values related to the most visited graph because of the flatness that characterizes the posterior probability of the visited graphs. In Figure 4.5 we report the precision matrix we used to generate the data and its pointwise estimates. Both of them show only the off-diagonal elements, the diagonal values were removed for plotting reasons. Indeed, to guarantee the matrix to be positive definite, the GWishart distributed matrices are characterized by diagonal elements whose values are much higher than the off-diagonal

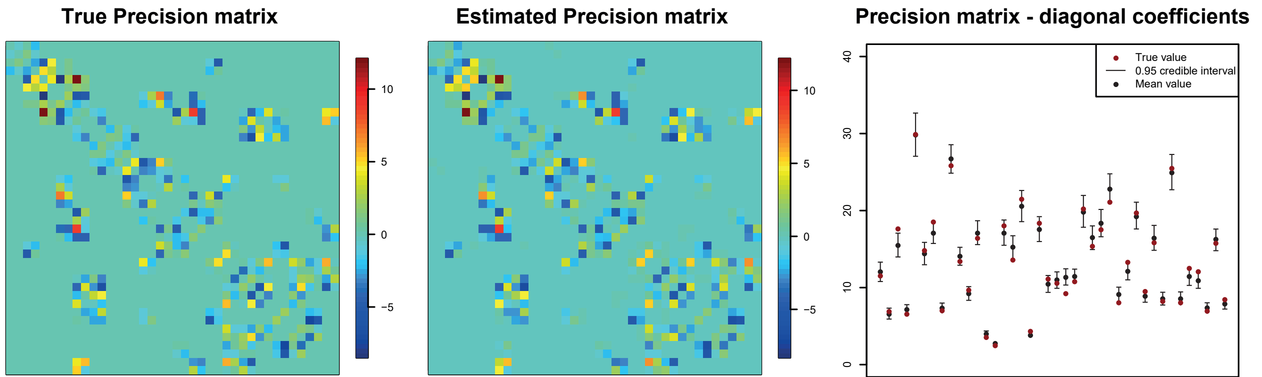

Figure 4.5. The true precision matrix (left panel) and its posterior estimate (middle panel). For plotting purposes, both are represented without the diagonal elements, which are instead reported in the rightmost panel.

ones. It is then convenient to plot them without those values, otherwise the underlying structure can not be seen properly. A summary of the diagonal elements is shown in the rightmost panel. We can conclude that our solution is also able to provide a fair estimate of the precision elements, which is also confirmed by a low value of the Kullback-Leibler divergence, $\mathrm{KL}\left(\boldsymbol{K}_{\text{true}}, \hat{\boldsymbol{K}}\right) = 0.3353$.

The test performed in Figure 4.3 is not a isolated example but we decided to illustrate it as a meaningful representative of a recurrent situation. We indeed repeated the same experiments for 15 different datasets: the true underlying graphs were randomly generated using different sparsity indices uniformly distributed in $[0.2, 0.6]$. The true

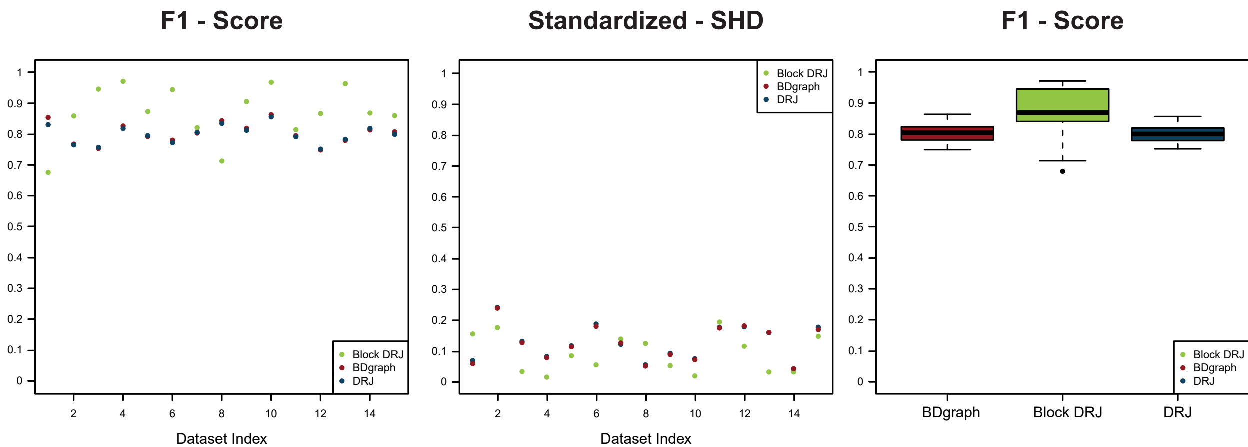

Figure 4.6. Analysis with high dimensional graphs, $p = 40$, multiple times and using graphs having different sparsity indices. For each dataset, we reported the F_1-scores (left panel) and the Standardized-SHD values (middle panel). The rightmost panel represents the boxplot for the F_1-scores.

precision matrices were drawn from a $\mathrm{GWishart}\left(3, \boldsymbol{I}_p\right)$. The resulting F_1-scores and Standardized-SHD values are reported in Figure 4.6 and we see that they reflect the

situation previously described: most of the times the Block DRJ performs better than `BDgraph` in terms of both indices, moreover the Birth and Death solutions are comparable to ones obtained with the `BGSL` implementation of the DRJ. The median F_1-score of the Block DRJ is around 0.85, but, if we look at its boxplots, we see that the Block version is wider than the complete counterpart. The low tail is stretched by a couple of repetitions, denoting a bit of instability. However in general it is the one providing the best estimates.

- **Experiment 2 - 40 nodes and different values of observations**
  The setting of this second experiment is similar to the previous one. We still use simulated datasets, the dimension of the graph is $p = 40$ and the number of groups is 20 as before. We used different graphs with respect to the previous experiment to be sure that the that was not a fortunate example. The peculiarity of this experiment is that, keeping p fixed, we vary the number n of observed data used in the estimation process. Graphical models are indeed often applied to infer complex relationships among large numbers of variables with a relatively small number of observations (Mohammadi and Wit 2015), even in the large p small n framework (Bhadra and Mallick 2013). Figure 4.7 displays the F_1-scores and Standardized-SHD values. We reported the n/p ratio on the x-axis .
  We can see that as n increases, the estimates become sharper and sharper, which

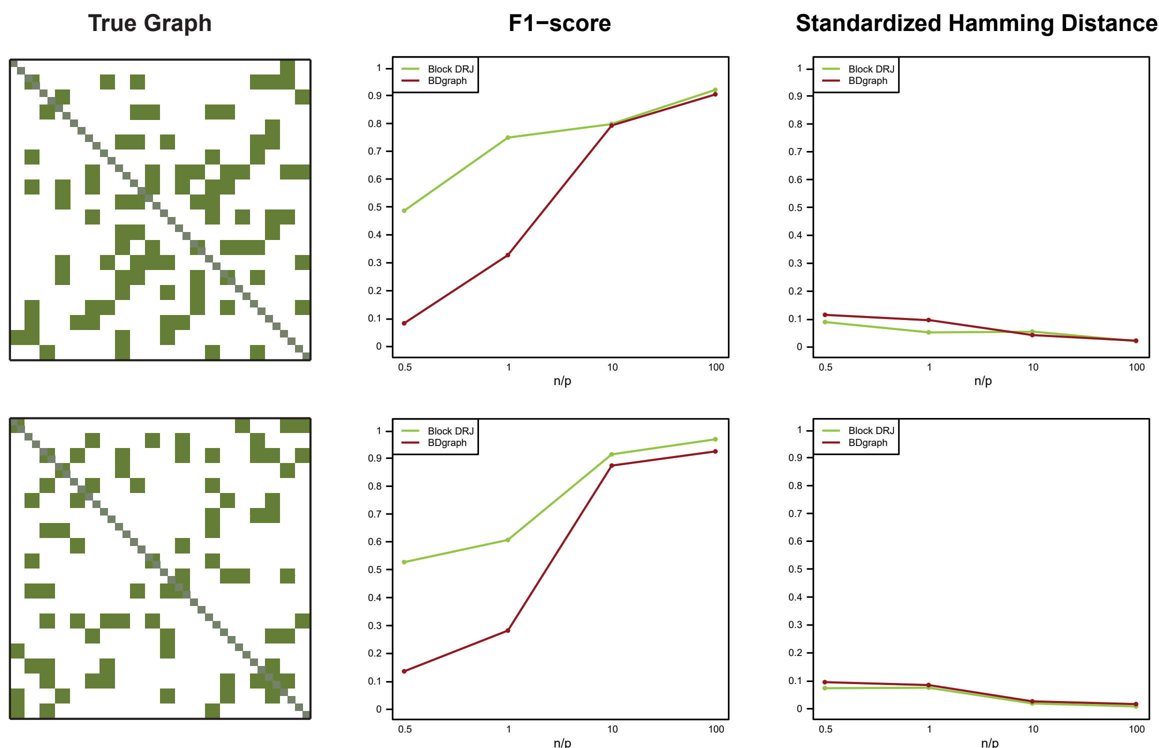

Figure 4.7. Two experiments to assess the performances when n increases. On the x-axis we report the n/p ratio.

assesses the convergence of the implemented method. It is not the only thing to be noted, indeed even when the ratio in smaller or equal than 1, we are able to partially recover the structure of the graph. In particular, `BDgraph` is outperformed in those two situations, at least in terms of the F_1-score, while that difference is not capture by the Standardized-SHD. This is indeed one of those cases when the second index is deceived by the sparsity of the underlying graph which is equal to 0.23 in the first scenario and 0.19 in the second one.

Looking at the confusion matrices when $n/p = 0.5$, see Table 4.3, we note that the `BDgraph` estimate is forcing almost all edges to be null. Since the number of zeros is therefore almost correct, the overall number of misclassified links is rather low which may lead to believe that the resulting graph is actually close to the real one. This is misleading because the number of actual discoveries is instead very low, around 4% in the first example and 7% in the second one. The Block DRJ is much better in this sense, around 36% in the first case and 43% in the second one.

Block DRJ		Estimated Graph		BDgraph		Estimated Graph	
		0	1			0	1
True Graph	0	581	20	True Graph	0	597	4
	1	115	64		1	171	8

Table 4.3. Confusion matrices referring to the first experiment displayed in Figure 4.7 in the case $n/p = 0.5$.

- **Experiment 3 - 30 nodes but incomplete blocks**

  In the previous experiments, we always set the true underlying graph to have an exact block structure, or to be more precise, given the groups, it was a draw from a truncated Bernoulli prior, see Section 1.2. This situation is not realistic because of our limiting hypothesis that the nodes within each block have to be fully connected or not linked at all. We made such a choice to reduce the dimensionality of the graph space and to induce a more interpretable structure in the adjacency matrix, but it is just a modelling hypothesis and it is unlikely that a real phenomenon is generated in this way. A more realistic situation could be such that variables do actually interact in groups, but they are not all necessarily connected one another.

  We try to replicate this situation in the following way: first we draw a block structured graph from a truncated Bernoulli prior having sparsity parameter equal to 0.2. Then, within each included block, each edge can be removed with probability equal to 0.25. By means of this procedure, we create a sparse graph with incomplete blocks such that the block structure can still be guessed. An example of a graph obtained in this way is represented in Figure 4.8.

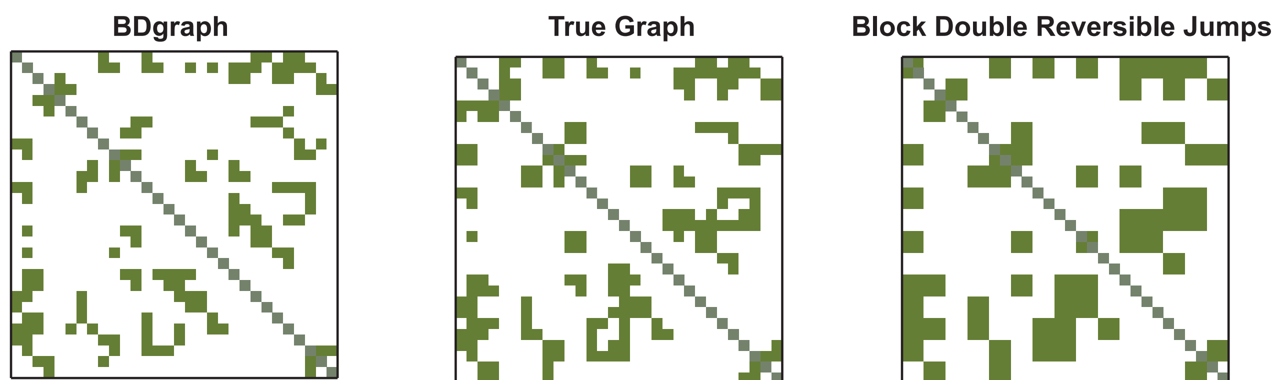

Figure 4.8. The real underlying graph (middle panel) has not an exact block structure. The Block DRJ estimates (right panel) is more inclined to include anyway those blocks, leading to some false positiveness. On the other hand, the `BDgraph` estimate (left panel) is rather conservative and therefore has an higher number of false zeros.

Figure 4.8 reports a significant experiment for a graph having $p = 30$ nodes. We repeated the same analysis, using the same observed values, both by means of `BDgraph` and the Block DRJ, choosing as usual the number of groups to be equal to $p/2$. The respective confusion matrices are reported in Table 4.4.
A simple visual inspection suggests that our approach is more inclined to include incomplete blocks rather than to discard them, which is exactly what we expected and that we hoped to see. If we had observed the opposite behaviour, which is the inclusion of a block only if it is complete, we would have ended up with an almost empty graph, no discoveries would have been done and it would have been a terrible solution.
However, this kind of solution necessarily implies a certain number of false discoveries but this is coherent with our modeling hypothesis. We already took into account that, even when we use the most sophisticated technique available right now, some links are misclassified. In a more realistic scenario like this one, we do not aim to recover exactly the true graph but rather to give a more clear and defined view about the kind of relationship among the variables that are into play. For this reason, in this case we are not so much bothered by the 27 false positiveness, that are 0.23 of the overall entries estimated as 1s, a number that is indeed similar to the probability for an edge to be removed from a block when constructing the true graph (it was equal to 0.25). This means that those false 1s are due to the modelling hypothesis and not because of blocks placed where there should not be any link, as confirmed by a visual inspection of the solution.

Let us now look at the `BDgraph` estimate we obtained (see Figure 4.8, left panel). The implemented Birth and Death chain does not make any block modelling assumption, so in principle it should be able to recover the graph correctly. In practice, as soon as the

dimension of the graph is not trivial, this never happens. The tendency here is instead to be more conservative, which leads to a lower number of false discoveries, only 7, but a higher number of false 0s that necessarily implies also less true discoveries, 63 out of 90 with respect to the 87 done by the Block DRJ. The reason is that `BDgraph` is not looking for a structured graph and therefore it is reluctant to add a new link even if the closest ones are. In other words, edges are included independently and, since the method is thought to impose sparsity in the final estimate, many 1s are not recognized. To conclude, note that even if the number of misclassified links is similar, the Standardized-SHD is indeed equal to 0.038 for the Block DRJ and 0.043 for `BDgraph`, the F_1-score rewards the capability of our chain to be able to make more discoveries, its values are indeed 0.853 and 0.788, respectively. As usual, the efficiency indices are useful tools to guide the analysis but they need to be contextualized according to the specific application. In this case, it is true that we obtain a higher F_1-score but only the framework can tell us if it is fine to have a consistent number of false positiveness or if it is better to be more conservative.

BDgraph		Estimated	Graph
		0	1
True Graph	0	338	7
	1	27	63

Block DRJ		Estimated	Graph
		0	1
True Graph	0	255	27
	1	3	87

Table 4.4. Confusion matrices referring to the experiment displayed in Figure 4.8.

We carefully described the previous example beacuse it is a meaningful representative of a recurrent situation, indeed we now want to show that it was not an isolated case.

We repeated the same analysis but using different graphs, hence different datasets. Since it would be messy to report all confusion matrices, we synthesize the above-mentioned idea by means of the sensitivity $\left(\frac{\text{TP}}{\text{TP+FN}}\right)$ and specificity $\left(\frac{\text{TN}}{\text{TN+FP}}\right)$ indices, see Figure 4.9. We see that our solution is always better in terms of sensitivity since it performs better both in the number of discoveries (TP) and a low number false 0s. This means that it is unlikely that an edge estimated as missing is instead present in the real graph. On the other hand, the `BDgraph` solutions are better in terms of specificity, an included link signifies a high probability of the presence of an actual connection in the real graph. As already explained, this is due to the higher number of false positive in the `BGSL` solution. In the analyzed cases, specificity and sensitivity perfectly divide the obtained solutions, which is no longer true if we also look at the the F_1-scores and Standardized-SHD values. The latter, in particular, is very similar in the two cases. It means that the two approaches are equivalent in terms of misclassified values.

The take-away message of this analysis is that when dealing with a realistic structured

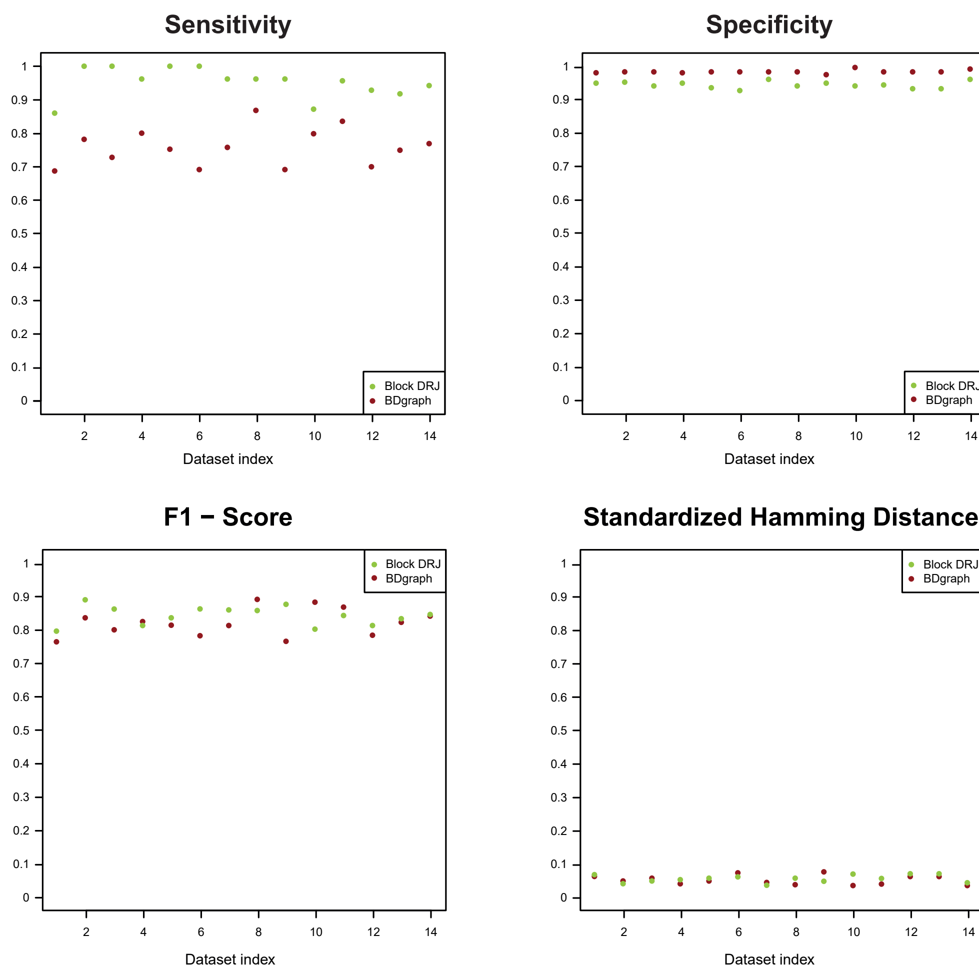

Figure 4.9. The analysis with incomplete graphs has been repeated 14 more times. We here report the Sensitivity (up-left panel) and the Specificity (up-right panel) indices as well as the F_1-scores (down-left panel) and the Standardized-SHD (down-right panel). The first two perfectly separates the estimates obtained with `BDgraph` (red bullet) and Block DRJ (green bullet) while for the other ones the separation is not neat.

graph with incomplete blocks, the expected findings from a search in the complete graph space or just in the block one are comparable. There is not an absolute criterion stating that one is better than the other, but we were able to identify a significant difference in the kind of solutions we get. What is the best approach to be used depends on the context, if one has to be conservative or if it is fine to admit some false discoveries, if sensibility is to be preferred to specificity or viceversa. Another aspect we did not explicitly consider in the previous examples but that it is obvious is that the estimated matrix has a much more precise and defined structure. As the number of nodes grows, a structured solution is much more meaningful that a sparse one, as it reveals more information about the underlying phenomenon that generates the observed data from a wider point of view.

- **Experiment 4 - noisy setting**

  Inspired by a simulation study presented in Kumar et al. (2020), we aim to learn a graph and precision matrix under a noisy setting. At first, we generate the real underlying graph G and precision matrix $\boldsymbol{K}_{\text{true}}$ as we did in experiment 1, then we add random noise. Specifically, we consider a scenario where the data $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$ used to feed the `GGMsampler` are drawn from $\boldsymbol{y}_i \sim N_p\left(\boldsymbol{0}, \boldsymbol{K}_{\text{noisy}}\right)$, $i = 1, \ldots, n$ where $\boldsymbol{K}_{\text{noisy}}$ is a random perturbation of $\boldsymbol{K}_{\text{true}}$ generated as follows: we start setting $\boldsymbol{K}_{\text{noisy}} = \boldsymbol{K}_{\text{true}}$, then each possible value, including the ones on the diagonal, has probability equal to s to be perturbed by adding a random noise ηu, with $u \sim \text{Unif}\left(-k^*, k^*\right)$, $\eta = 0.1$ and $k^* = \max_{i<j}|k_{ij}^{\text{true}}|$. Figure 4.10 gives a visual representation of the graph we chose for this experiment, the true precision matrix $\boldsymbol{K}_{\text{true}}$ and an example for $\boldsymbol{K}_{\text{noisy}}$. To assess the performance of Block DRJ in this

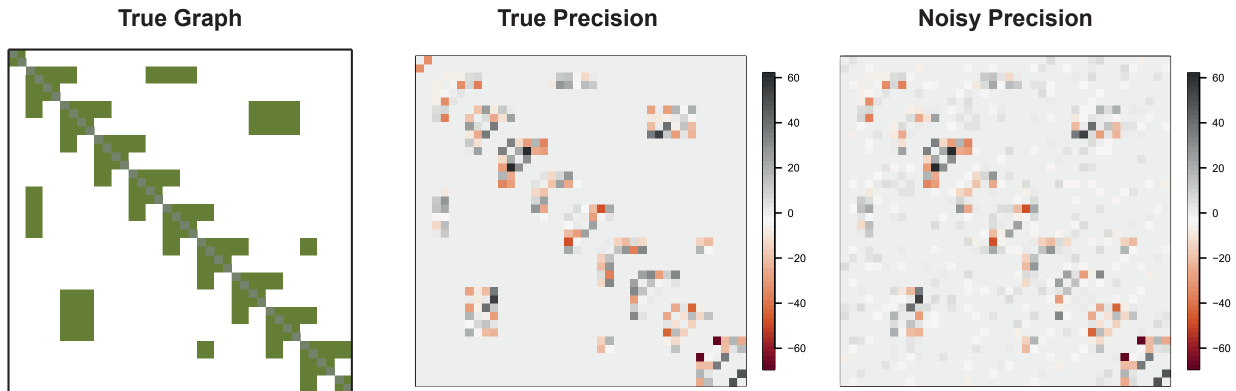

Figure 4.10. The leftmost panel displays the true graphs used to generate the true precision matrix $\boldsymbol{K}_{\text{true}}$, which is shown in the middle panel. The data are then generated using $\boldsymbol{K}_{\text{noisy}}$. An example of the latter is presented in the rightmost panel, it has been generated setting $s = 0.25$. For plotting purposes, we removed the diagonal in both matrices.

  setting, we test it under different volumes of noise, in particular for $s = 0.10, 0.20, 0.25$. We always maintain the same graph G and the same $\boldsymbol{K}_{\text{true}}$ we showed in Figure 4.10, the only difference between different experiments is the level of noisy. Moreover, for each s, we repeat the analysis multiple times (15), generating every time a new $\boldsymbol{K}_{\text{noisy}}$. Figure 4.11 shows the indices of interest.

  Block DRJ outperformed `BDgraph` on every datasetes, both in terms of F_1-scores and Standardized-SHD. Its robustness is due to the fact that a single, isolated link is not enough to conclude that an whole block has to be inserted in the final graph. Those lonely values are not compatible with the block structured graph that Block DRJ is looking for, therefore they are rightly ignored. On the other hand, `BDgraph` does not look for any particular structure, hence it does not recognize the perturbed values as noise. As already noticed in the previous experiments, the F_1-scores is the index that

best highlights the difference between the two methods.
If we compare those boxplots with the noisy-free setting, see experiment 1, Figure 4.6, as expected, we do register lower scores characterized by a decreasing trend as the noise volume increases. However, the performance plummet for Block DRJ is not as dramatic as for `BDgraph`.
Finally, Figure 4.11 also reports the Kullback-Leibler divergence for the estimated precision matrix with respect to $\boldsymbol{K}_{\text{true}}$, $KL\left(\boldsymbol{K}_{\text{true}}, \hat{\boldsymbol{K}}\right)$. Not surprisingly, Block DRJ outperforms `BDgraph` because it is better in filtering out the noise.

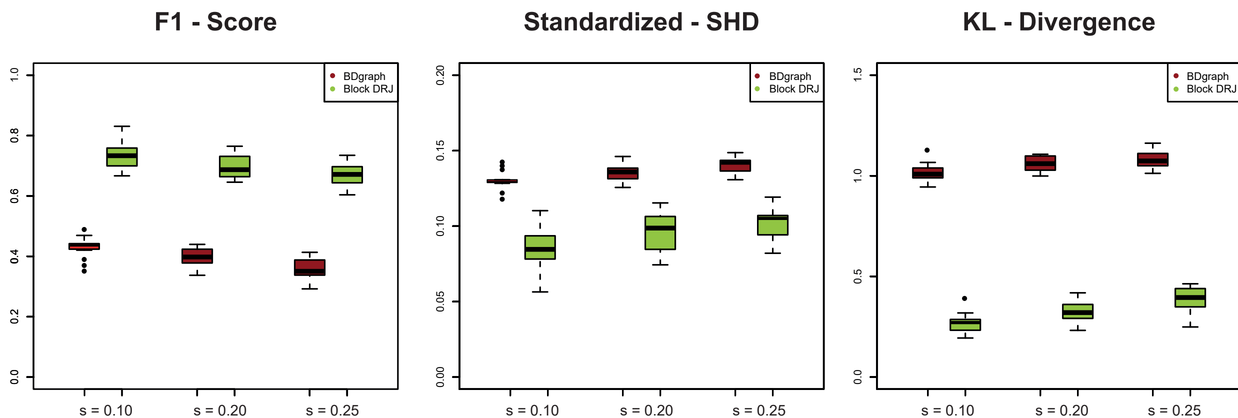

Figure 4.11. Boxplots for F_1-scores (left panel), Standardized-SHD (middle panel) and Kullback-Leibler divergence (right panel). The same datasets has been used for Block DRJ (green) and `BDgraph` (red).

- **Experiment 5 - 30 nodes and 10 groups**

  In the experiments above, we always set the number of groups to be equal to half of the number of vertices, which imply off-diagonal blocks of size 2×2. When the size of the groups increases, let us say $n_groups = p/3$, the acceptance rate of the chain dramatically plummets. The reason is that in this scenario, to modify an off-diagonal block we need to perturb 9 free-elements of $\boldsymbol{\Phi} = Chol(\boldsymbol{K})$, which in turn implies additional changes among the non free-elements. One of the computational problems is that the variance assessing the magnitude of this perturbation, which is called σ_g^2 in Chapter 2, is the same for all those entries and it is fixed by a naive tuning procedure. We performed some tests by fixing $p = 30$ and 10 groups. The acceptance rate was dramatically low, around 0.58%. We tried to face the problem by running very long chains and performing some thinning. We repeated the analysis using different graphs and different datasets, the boxplots for the F_1-scores and the Standardized-SHD values are displayed in Figure 4.12. We can see that most of the times we obtain values comparable with respect to the previous experiments. A couple of time we even succeeded in recovering exactly the true graph. Unfortunately the method presents also some instabilities, comparing Figure 4.12 and Figure 4.6 we note that the median lines for the

F_1-scores have similar values, both slightly above 0.8, but in this second case the boxplot is much larger, in particular the low tail is wider which denote that some estimates were not precise at all.

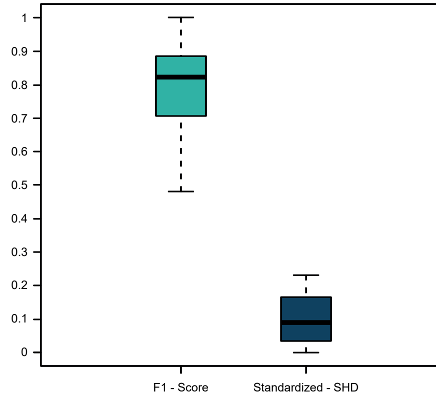

Figure 4.12. F_1-scores (left boxplot) and Standardized-SHD values (right boxplot) for an experiment where $p = 30$ and $n_groups = p/3$.

4.3 Testing FLMsampler

This section is dedicated to the validation of the `FLMsampler`, the only one among the `BGSL` samplers that do not perform any graphical search. Since we do not need to worry about the estimation of the graph, we can exploit this simpler scenario to properly introduce the functional framework, to validate our Gibbs sampling strategy and to show what is the impact of the graph on functional data.

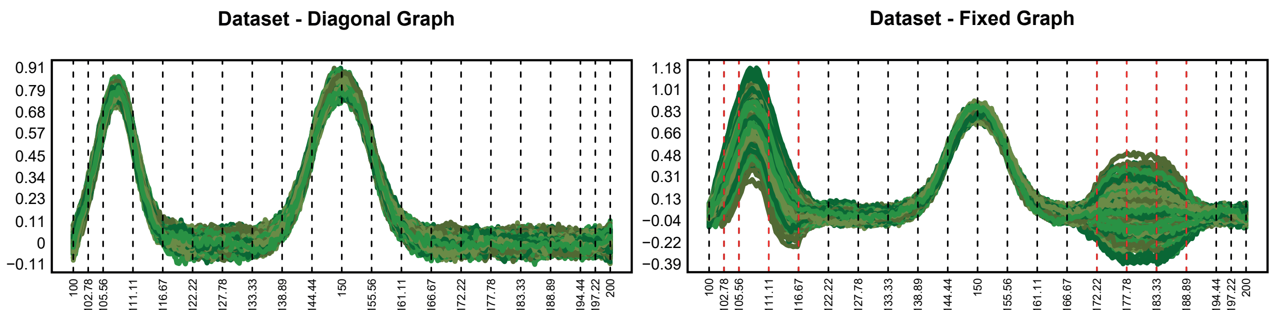

Figure 4.13. Two functional datasets. Curves on the left are generated from a classical smoothing model, see (3.15), no graphical part is involved. The second setting (right panel) is instead obtained from model (3.16) where the underlying graph has only one off diagonal block of size 3×3 connecting the bands highlighted in the plot. The Gaussian noise affecting the measurements is the same in both cases, the additional noise one can note is due to such dependencies described by the graphical part.

After choosing $p = 20$ B-spline basis, we simulated $n = 500$ noisy curves in the domain $[100, 200]$ from model (3.15), which is the one not affected by the graph meaning that within

each curve the p spline coefficients we called $\boldsymbol{\beta}$s are independent one another. The Gaussian noise associated with each observation has zero mean and precision equal to $\tau_\varepsilon^2 = 5000$ so that all curves are reasonably similar. Regarding the drawn of the $\boldsymbol{\beta}$s, $\boldsymbol{\beta}_i \sim N(\boldsymbol{\mu}, \boldsymbol{K})$, for each $j = 1, \ldots, p$ and $i = 1, \ldots, n$ where the $\boldsymbol{\mu}$ coefficients are generated in such a way that all curves present two well-recognizable peaks. For the first dataset, the precision matrix is set to be diagonal, $\boldsymbol{K} = \text{diag}\left(\tau_1^2, \ldots, \tau_p^2\right)$ where the coefficients τ_j^2 are normally spread around 1000, for each $j = 1, \ldots, p$.
The resulting data are displayed in the left panel of Figure 4.13, the dotted vertical lines are meant to highlight the intervals where a particular spline dominates all the others and it can therefore be represented by one of the p $\boldsymbol{\beta}$ coefficients. See Section 2.5 for more details.
We then simulated n additional curves from model (3.16) to show the effect of dependent spline coefficients. We recall that in this case, the graph is fixed but not empty and therefore the precision matrix is modelled a priori as a GWishart. In particular, we chose a graph that presents only one off-diagonal block to connect the splines indexed by $2, 3, 4$ with $15, 16, 17$. The corresponding precision matrix has the same diagonal elements we used for the previous dataset and the entries in the new block are all equal to 330. The new curves are reported in the right panel of Figure 4.13, where we also highlight those bands that are associated with the graph, which indeed present some extra variability with respect to the previous case. The number of grid points is equal to 250 in both cases.

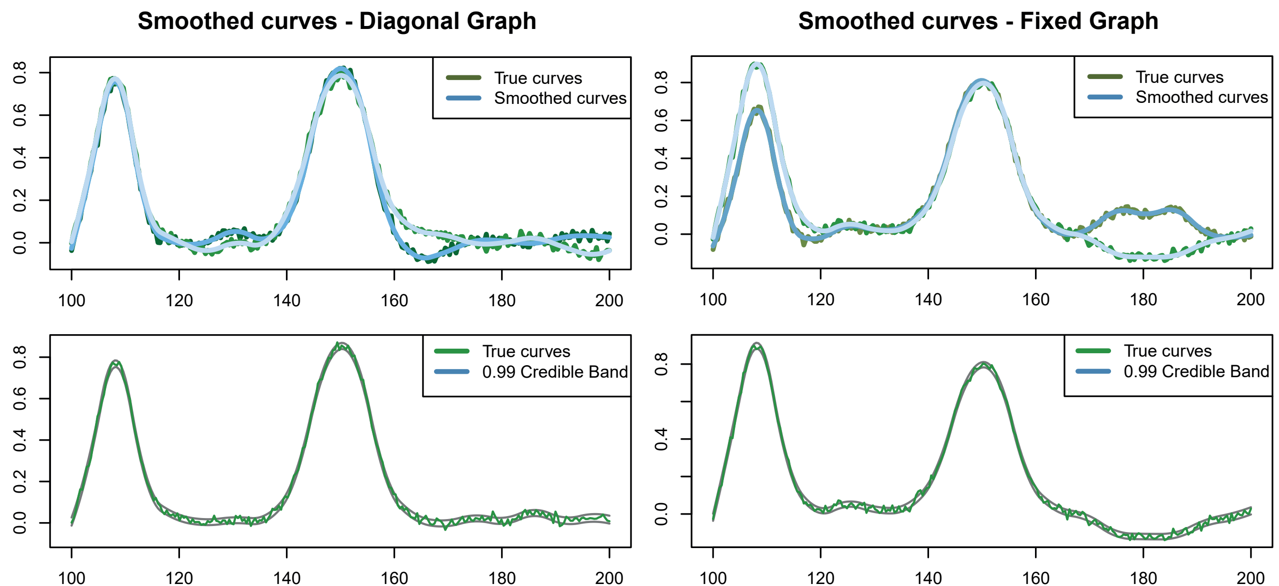

Figure 4.14. A couple of smoothed curves for each dataset (upper panels) as well as the 99% credible bands (lower panels).

For the sampling procedure, we fixed the hyperparameters to obtain sufficiently vague priors but also taking into account that the data are rather regular, which means that we expect high values for the precision coefficients. We indeed choose $\sigma_\mu^2 = 100$, $a_\varepsilon = 25$, $b_\tau = 10$

and $b_\varepsilon = b_\tau = 0.01$ while in the second setting, we used a $\mathrm{GWishart}(3, 0.01\boldsymbol{I}_p)$ for the precision matrix.
Since there is no graph to be estimated, we can rely on shorter chains with respect to the previous section, we use 8000 iterations as burn-in period followed by 8000 more but with a thinning value of 4, which implies a total number of 2000 iterations saved. Runtime is about 20 seconds in both cases.

Figure 4.14 shows a couple of smoothed curves for each dataset as well as the 99% credible band. A simple visual inspection suggests an high quality smoothing, the regular curves faithfully follow the trend of the observed ones but removing the noise. The reliability of the pointwise estimates is insured by the thinness of the credible intervals. For plotting reasons, we only reported a few curves but the same conclusions apply to all others.

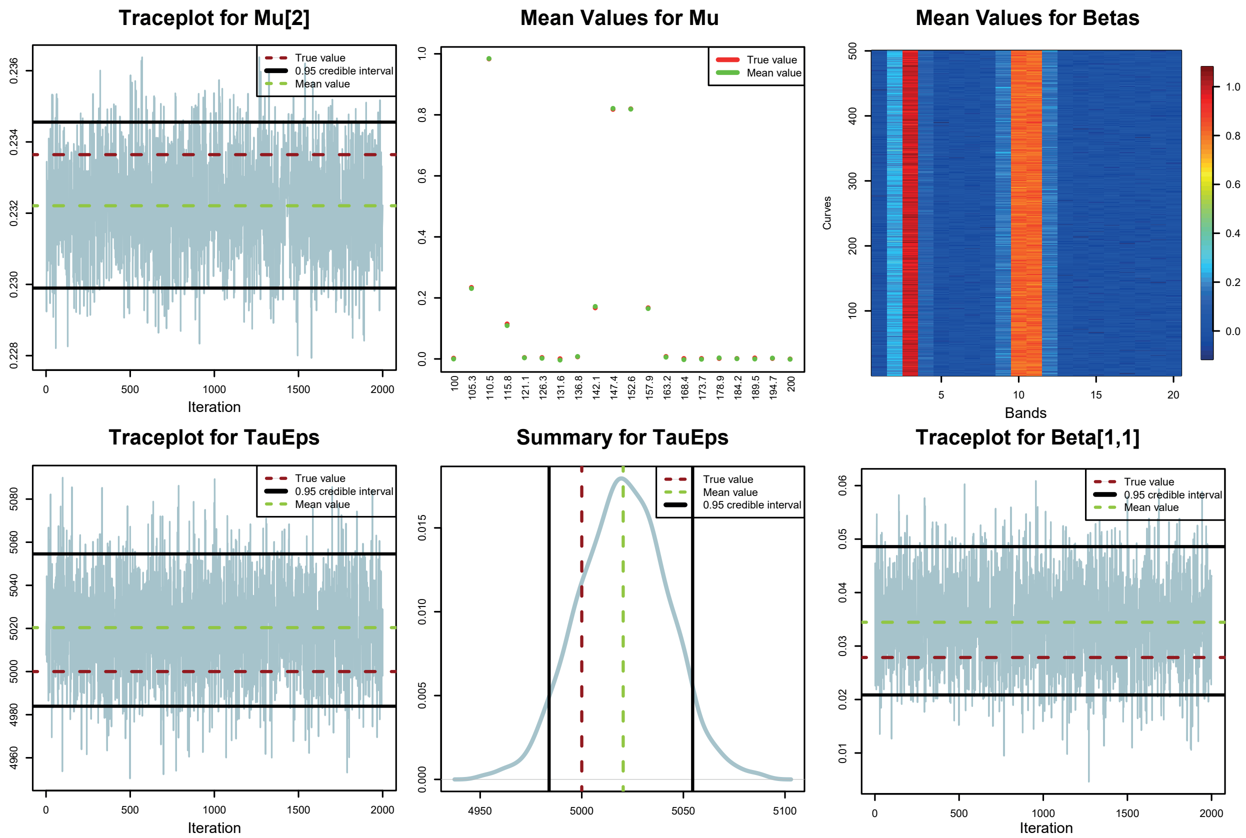

Figure 4.15. Some diagnostic plots for the regression parameters of `FLMsampler_diagoanl`.

Figure 4.15 is a summary of parameters $\boldsymbol{\beta}$, $\boldsymbol{\mu}$ and τ_ε^2. They all refer to the first dataset but the same considerations are also valid for the second one. The traceplots suggests that all chains reached convergence and none of them present trends to be worried about. They are thick, stable, the true underlying value is reasonable close to the median one and it always fall within the 95% credible interval. The estimates for the $\boldsymbol{\mu}$ coefficients are so tight that it is difficult to distinguish the real values from their estimates. We did not plot their credible intervals because they are so short that they would be almost impossible to see, for example

look at the one reported for μ_2. Similar considerations hold for τ_ε^2, for which we also plotted the estimated density.

The final Figure 4.16 contains a summary of the coefficients of the precision matrix $\boldsymbol{K}$. In the first experiment it is modelled as $\boldsymbol{K} = \mathrm{diag}\left(\tau_1^2, \ldots, \tau_p^2\right)$, in the leftmost panel we plotted the pointwise and interval estimates for each τ_j^2. In the second case the only non null elements are the diagonal ones, summarized in the middle panel, and one off-diagonal block of size 3×3 having all values equal to 330, whose estimation is showed in the rightmost plot. The credible intervals looks wider than the ones in Figure 4.15, the reason probably is that the variance they are catching, the one within each curve, is in same sense masked by the one explained by τ_ε^2. Since the latter is directly related to observed data, its estimate is more precise than the one explained by such parameters, which, instead, describe the variance of unobserved quantities. Nevertheless, the pointwise estimates are most of the times close to the real values, which usually fall inside the 95% credible intervals.

One last particular to be noted is that the estimates obtained by means of the GWishart prior are as reliable as the simpler diagonal case. The credible intervals for the diagonal elements show a similar behaviour in the two cases and the off-diagonal elements are sufficiently well estimated. To conclude, in the second setting we have $\mathrm{KL}\left(\boldsymbol{K}_{\mathrm{true}}, \hat{\boldsymbol{K}}\right) = 0.0241$.

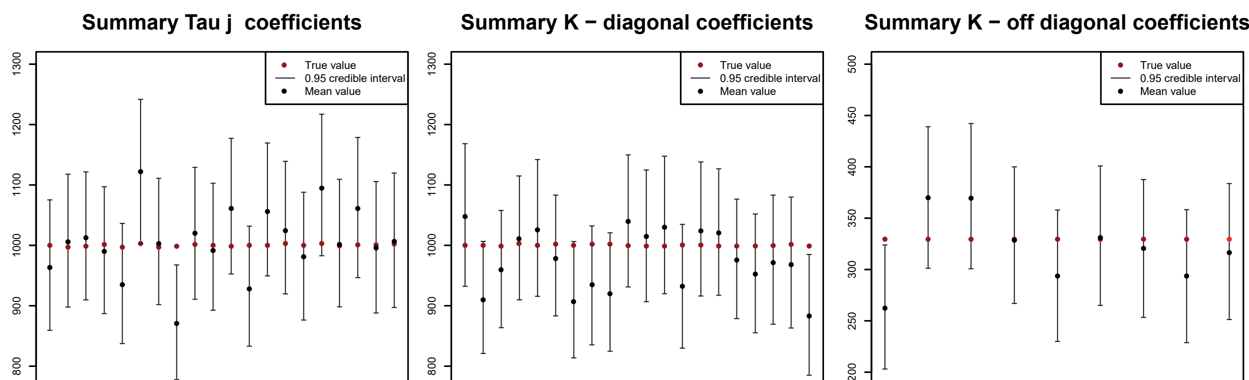

Figure 4.16. Summaries for the precision matrix $\boldsymbol{K}$. The first dataset was generated by modelling such matrix as diagonal $\boldsymbol{K} = \mathrm{diag}\left(\tau_1^2, \ldots, \tau_p^2\right)$, the leftmost panel displays pointwise and interval estimates for the τ_j^2 coefficients. For the second dataset, the only off diagonal elements of $\boldsymbol{K}$ that are not null are a 3×3 block, whose values are summarized in the rightmost plot. The middle one is instead about the diagonal elements in this second experiment.

The previous examples were meant to validate the correct implementation of the two samplers. The next question we want to answer is what is the difference between them, how does the graphical part improve the smoothing strategy? We already discuss the advantage of discovering patterns and dependencies between different portion of the domain. Even though that is our primal interest, it may be considered as a byproduct of the smoothing procedure, we would also like to visualize the impact that such discoveries have on the quality of the

smoothing.
To answer such a question, we consider an example when we have to deal with missing values, or more specifically, when some curves are not measured in every point of the grid. Consider the same dataset on the right panel of Figure 4.13, the one with the true underlying graph that is not diagonal. We recall that such curves are generated from $r = 250$ measurements on an uniform grid. However this time we do not use all available data for the smoothing: we take 490 complete curves, i.e using all the r data but for the remaining 10 we exclude the values in the range $[170, 180]$.

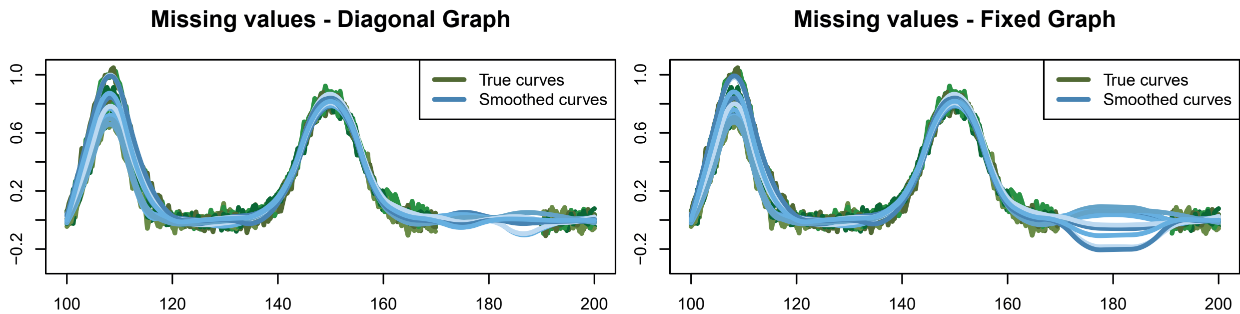

Figure 4.17. Experiments with missing points, we did not use any of the points in the range $[170, 180]$. Modeling the graph as diagonal (left panel) the resulting curves remain flat, the long term interaction given by the off-diagonal graph (right panel) is instead able to inflate the smoothed curves.

No surprises happen for the 490 complete curves when smoothing them by means of both version of the `FLMsampler`, both methods are able to recover a faithful regular version of the real ones. Figure 4.17 shows how different are the shapes of the 10 incomplete incomplete functional data. Since we have some missing values, each model tries to fill that portion by borrowing information from the other coefficients but (3.15) does not take into account any dependence and therefore we end up having flat curves that are not able to catch the real pattern. The second model, (3.16), thanks to the information shared by the graph, is instead able to guess the right behaviour, looking at the right panel of Figure 4.17 we indeed see that are not flat by they inflate a bit. We underline that this behaviour is due to long-term interactions, to capture the shape in the segment $[170, 180]$ we mainly use information we get from the first peak in $[100, 120]$. This feature is totally new in the smoothing procedure, which instead exploits only local interactions (Ramsay and Silverman 2005).

This experiment has been tailored to visualize the effect of a single off-diagonal block in the graph, the missing points were indeed placed exactly in a region that was much influenced by another portion of the domain, it is not the most general case but we belive it may be a reference example to show how graphical models improves can improve the smoothing technique by removing the independence hypothesis. In this experiment, we kept the graph fixed, the last point to be addressed is if the `FGMsampler` is are able to estimate it.

4.4 Testing FGMsampler

We conclude the validation experiments by giving a couple of examples of the `FGMsampler`. We test it both in low and high dimension, in the first case we fix the number of basis functions to $p = 20$, in the second we use $p = 40$. In both cases, the precision matrix $\boldsymbol{K}$ was drawn from $\text{GWishart}(3, 0.1\boldsymbol{I}_p)$. Curves are simulated adding more noise with respect to the example in

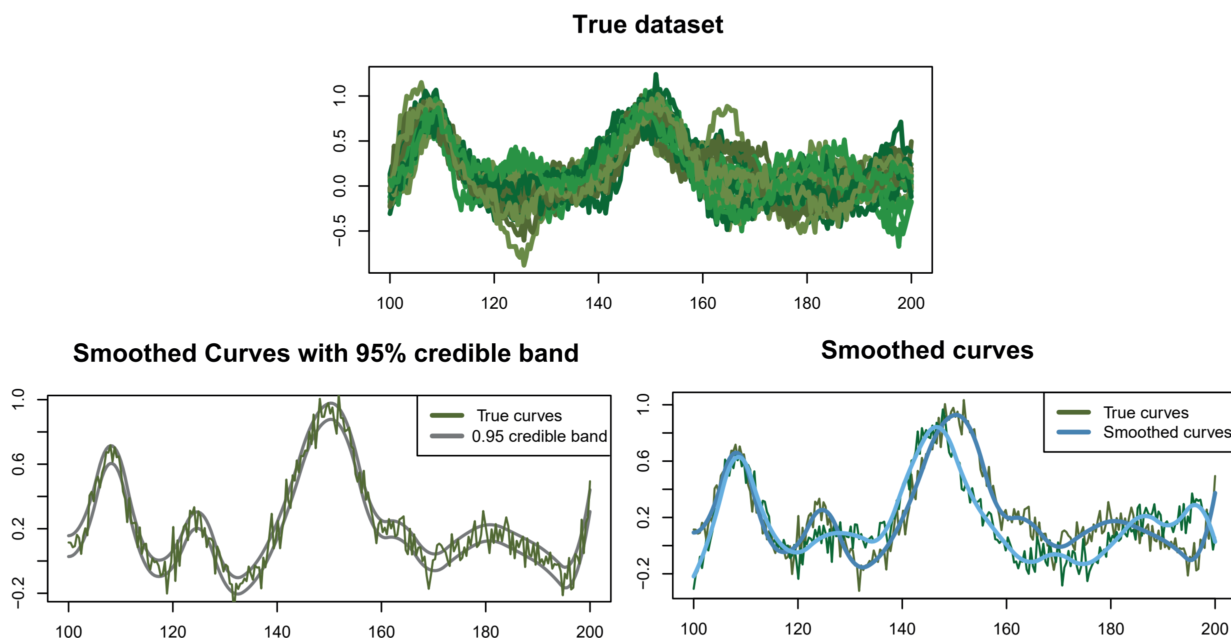

Figure 4.18. The noisy simulated dataset used to test the FGMsampler (upper panel) and the first two smoothed curves (lower right panel) and the 95% credible band for the first curve.

Figure 4.14, the true value of τ_ε^2 is indeed set to be 200. The resulting dataset, for $p = 20$, is given in Figure 4.18. We recall that this sampler has a double purpose; it performs a smoothing of the inserted noisy curves and estimates the graph which describes the relationship among parameters $\boldsymbol{\beta}$s.

This second task is more challenging with respect to the example shown in Section 4.2, indeed the graph is now set on unobserved quantities and not on data. The $\boldsymbol{\beta}$ coefficients have to be estimated as well, this implies additional uncertainty when we the search for the graph. We runned very long chains, 250′000 iterations plus 250′000 more as burn-in phase. We applied a thinning value of 10 for $\boldsymbol{\beta}$s, $\boldsymbol{\mu}$ and τ_ε^2.

The quality in the estimation of those variables related to the smoothing procedure, i.e the $\boldsymbol{\beta}$s, $\boldsymbol{\mu}$ and τ_ε^2, is very similar to what we obtained using the `FLMsampler`. The pointwise estimates are reasonably similar to the true values and the length of the credible intervals is usually very short, giving us reliable estimates. This is true both for the $p = 20$ and the $p = 40$ case.

We do not report diagnosis plots because they would be a repetition of what we already showed

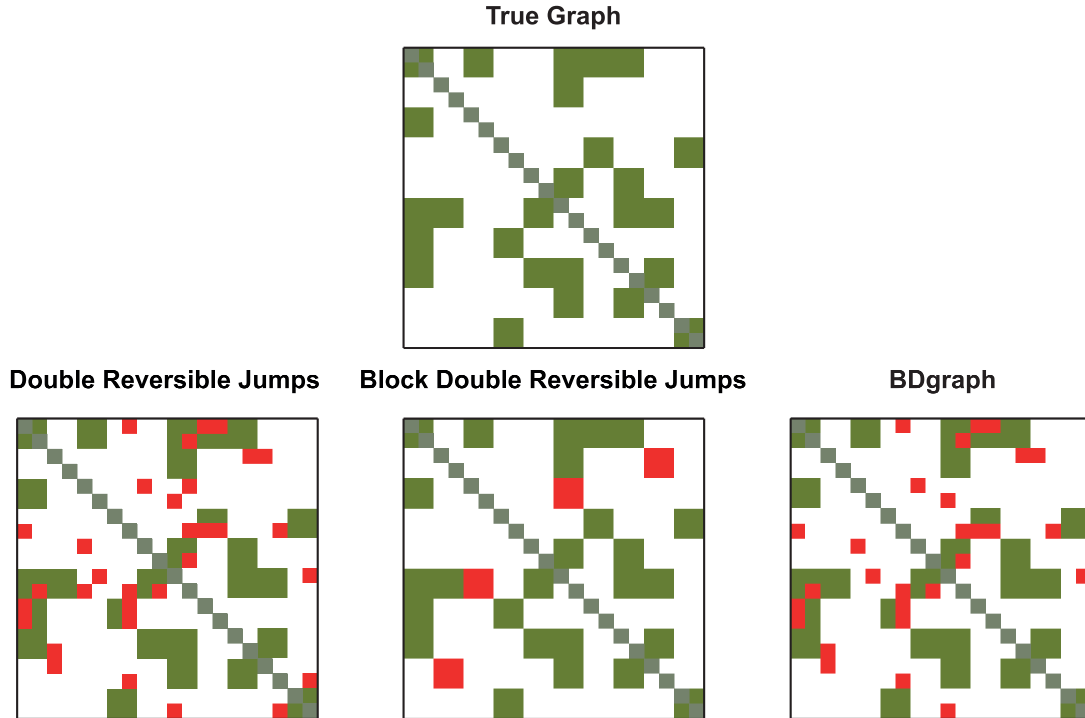

Figure 4.19. The true underlying graph (upper panel) and the estimated graphs. The left-hand panel and the middle one were obtained using BGSL package, the right-hand one using BDgraph. Green squares represent the included links, the red squares stand for edges that are wrongly estimated. We graph size is defined by $p = 20$.

in the previous section. More in general, we may say that for those quantities we do not expect significant differences if we use a `FLMsampler`, which keeps the graph fixed, or a `FGMsampler` that also has to estimate the graph.
For the $p = 20$ case, Figure 4.18 also presents a couple of smoothed curve and the 95% credible band for the first curve. The same level of precision is observed for all other curves, also in the $p = 40$ case.

The most interesting feature of the `FGMsampler` is the analysis of the graphical part. We first present the low dimensional case.
To provide an exhaustive validation, we repeated the same experiment, using the same dataset but using three different graphical methods: the Double Reversible Jumps approach of `BGSL`, tested both in its complete graphs version (denoted by DRJ) and its block structured graph version (Block DRJ), is compared to the Birth and Death approach of `BDgraph`. The final estimates are presented in Figure 4.19.
Even if the graph is now used to describe the relationship structure of the $\boldsymbol{\beta}$ coefficients, we are still able to identify the correct structure in all three cases and the differences among them show a very similar behaviour to the one observed in Section 4.2. As before, the most precise one is the *Block Double Reversible Jumps* which wrongly includes in the final graph only 8 links out of 190, which gives us F_1-score = 0.926 and Standardized-SHD = 0.042. As already

noted in Figure 4.6, the DRJ and `BDgraph` arrive to very similar solutions, both in terms of F_1-scores (0.833, 0.851) and Standardized-SHD (0.084, 0.073). Looking at Figure 4.19 we also notice that, in this particular case, they misclassify the same links. Even if the way those methods navigate the space $\mathscr{G}$ of all possible graphs is totally different, in the end, they filter from the data very similar information and arrive to very similar solutions.

Figure 4.20 shows the traceplots of the size of the visited graphs for Block DRJ and DRJ; they

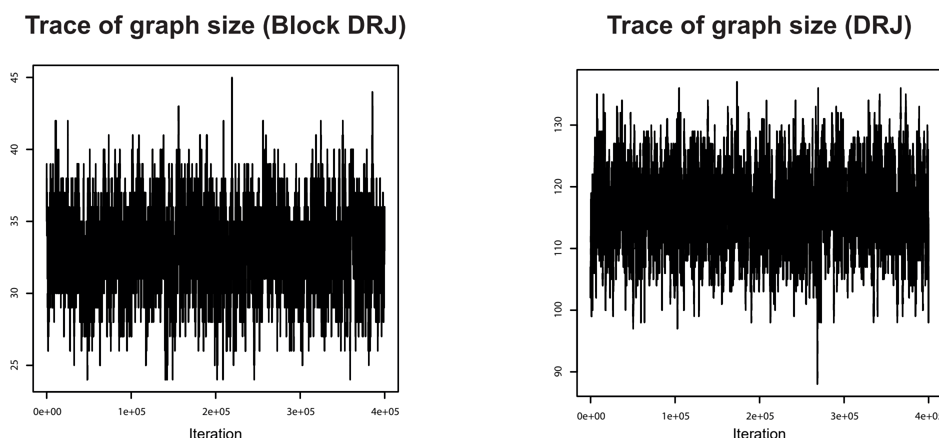

Figure 4.20. Traceplot of the size of the visited graphs, they are obtained via BGSL- Double Reversible Jump method for block graph (left) and complete graphs (right). The acceptance rate were respectively 12% and 40%.

asses the convergence of both chains which also swipe well the regions of interest. Moreover, we noticed that placing the GGM on the regression coefficients leads to a higher acceptance rate. For this example, it was near to 12% for block graphs and almost 40% for complete graphs.

All three final graphs were obtained by cutting the posterior probability of inclusion of each link to the threshold given by the *Bayesian False Discovery rate* criteria, see Section 3.5.4.

For the $p = 40$ scenario we only analyzed the Block DRJ approach, whose solution is given in Figure 4.21. Conclusions that regard `BDgraph` or DRJ are not different from the $p = 20$ case.

The relative indices are F_1-score = 0.948 and Standardized-SHD = 0.029, which are in line with the values we observed in Section 4.2 when the graph was used to describe the data. Figure 4.22 reports the true underlying precision matrix and the estimated one. As we did in Section 4.2, for plotting reasons, we first present the off-diagonal elements and the diagonal ones are shown in the last panel, along with their 95% credible intervals. A simple visual inspection suggests that the main structure is correctly identified, the heat maps are reasonably similar and all true values fall into the corresponding credible interval. The major difficulties are recorded for the estimation of the most extreme values, both for the diagonal and the off-diagonal elements. Such considerations are confirmed by the quantitative index, indeed KL = 0.957 which is small enough to conclude an high quality estimate.

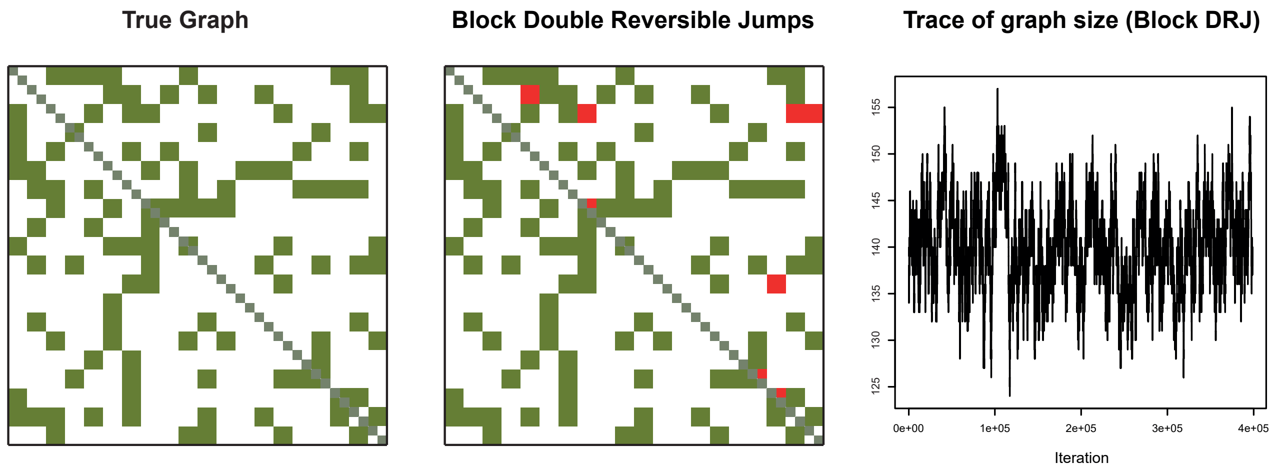

Figure 4.21. `FGMsampler` experiment with a 40 nodes graph. The Block DRJ estimate, shown in the middle panel, misclassify only 23 links out of 780 possibilities. The rightmost panel presents the corresponding traceplot of the sizes of the visited graphs.

Finally, Figure 4.21 also contains the traceplot of the size of the visited graphs. Even if the chain is not as thick as the $p = 20$ case, we can conclude that convergence was successfully attained and once again the acceptance rate was higher with respect to the tests of Section 4.2.

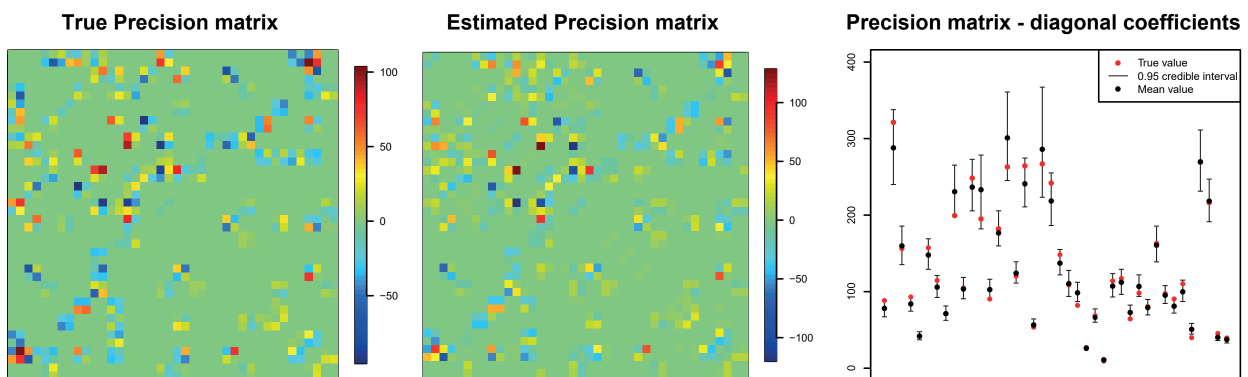

Figure 4.22. The true precision matrix (left panel) and its posterior estimate (middle panel). For plotting purposes, both are represented without the diagonal elements, which are instead reported in the rightmost panel. In both cases, our solution often underestimate the true values.

Wrapping up, first we assessed the performance of the `GGMsampler` in Section 4.2 by testing it in different scenarios, secondly Section 4.3 was dedicated to the `FLMsampler`, which does not involve any graph estimation, but it only focuses on the smoothing procedure of the functional data. Finally, we here presented a couple of tests done using the `FGMsampler` that combines the both techniques. Although the usage of the graphical part is rather innovative, our experience, supported by the two tests here reported, did not evidence particular differences with respect to the standard `GGMsampler` case. It may take a longer time for the chain to reach convergence

even if this issue is toned down by higher acceptance rates. Regarding the precision matrix estimation process, we do registered some instability with respect to the choice of the prior Inverse-Scale matrix $\boldsymbol{D}$. We consider it a fixed hyperparameter of the form $\boldsymbol{D} = c\boldsymbol{I}_p$, usually we set $c = 1$. The choice of the identity matrix is due to the lack of modelling information. What we experienced is that performances may be influenced by the value of c. In the future we may consider better modeling assumptions, for example using an additional scaling parameter GWishart $\left(3, \sigma_s^2 \boldsymbol{I}_p\right)$, where σ_s^2 is a priori distributed as a Gamma random variable. See Yang et al. (2014) and Yang et al. (2017) for further details.

4.5 Real data application - fruit purees dataset

Our work is motivated by the analysis of spectrometric data representing the absorbance spectrum of 351 fruit purees prepared using exclusively fresh whole strawberries, meaning that no adulterants where added. Data are publicly available at `https://data.mendeley.com/datasets/frrv2yd9rg`(Zheng et al. 2019) and reported in Figure 4.23. An exhaustive description can be found in Holland et al. (1998). We here limit ourselves to say that they were collected by means of *Infrared spectroscopy*, that measures the interaction of infrared radiation with matter by absorption, emission, or reflection. It is used to study and identify chemical substances or functional groups in solid, liquid, or gaseous forms. This technique is conducted with an instrument called infrared spectrometer, which produces the so-called infrared spectrum. For instance, mid-infrared spectroscopy coupled with chemometrics or statistical techniques have been used in literature to study the substances composition in food (Holland et al. 1998).

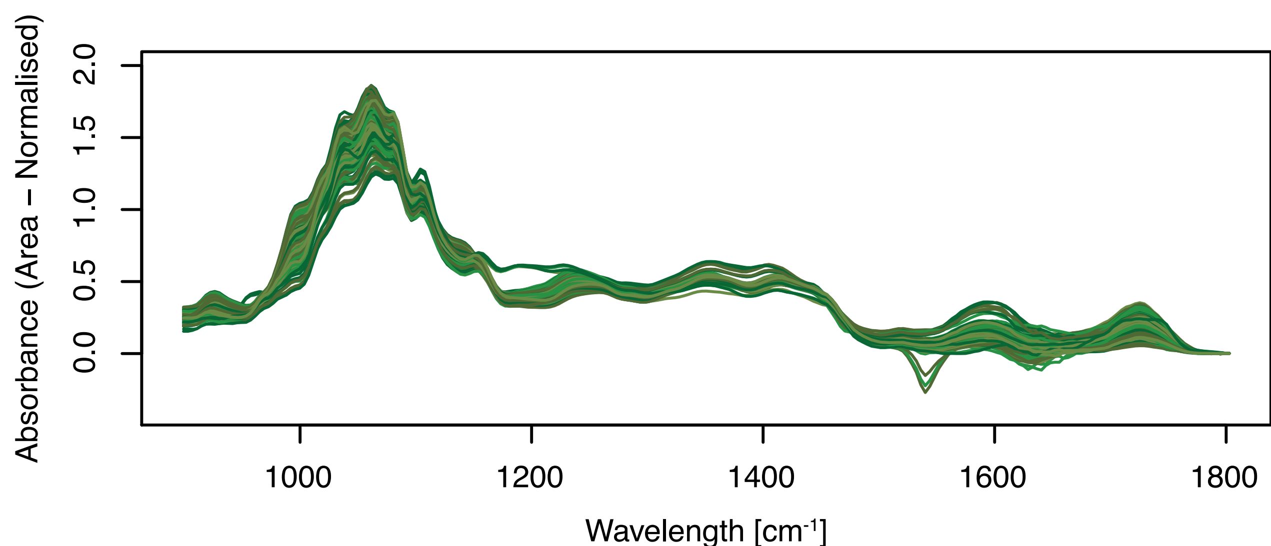

Figure 4.23. Plot of 351 spectra of absorbance of pure fruit purees, measured in 235 different wavelengths of the middle-infrared spectra.

The chemical analysis of such a complex spectrum is not trivial. If we observed the absorbance of a single, pure substance, we would see only one peak at a specific wavelength. As the object of the analysis becomes more elaborated, the spectrum itself becomes more difficult to be read because of many overlapping effects. In particular, those purees have a complex, heterogeneous composition which complicates the interpretation of its spectrum. The goal of our analysis is to provide useful insights about which wavelength bands are related to the different components of the purees, and more specifically about the dependency structure between wavelength bands. Since, from a mathematical point of view, a spectrum is a continuous function of the wavelength, our approach to the problem is to frame it within the functional graphical model setting we explained in Section 2.5. Indeed, the dependence structure between signals at different wavelengths is particularly informative in this sense: if two different bands of the spectrum are dependent, we can conclude that they refer to two closely related components or chemical groups.

The absorbance of each fruit puree sample was measured on an equally spaced grid of 235 different wavelength, whose range is $[899, 1802]$ cm^{-1}. The resulting spectra were then standardized with respect to the area under the curve, so that their final range is $[-0.1, 1.7]$. The shape of the spectra is usually very similar for all curves, they all exhibit a well-recognizable peak around the wavelength $1000 - 1200$ cm^{-1} and few secondary others, like the ones around 1600 cm^{-1} and 1700 cm^{-1}.

Similarly to the experiments performed in the previous section, we first fit model (2.50) using our Block DRJ approach to accomplish the graphical step of the sampling strategy, then we repeat the analysis by means of `BDgraph`. We use $p = 40$ B-spline basis functions, which as usual is a trade-off between good fitting of the smoothed curves and limited computational burden. In this case, the observed curves are not particularly noisy, implying that a good regular representation could have been accomplished also with a smaller p. We chose to use a larger number of bases functions because in the setting of spectrographic data a precise peak estimation is crucial.
As we did in Section 4.3, we fixed the hyperparameters to get vague prior distributions, $\sigma_\mu^2 = 100$, $a = 10$, $b = 0.01$ and a GWishart$(3, \boldsymbol{I}_p)$. We runned very long chains, $500'000$ iterations plus $300'000$ more as burn-in period. For the regression parameters $\boldsymbol{\beta}$s, $\boldsymbol{\mu}$ and τ_ε^2 we set a thinning value of 40. Figure 4.24 presents an example of two smoothed curves compared to the measured ones. Notice how they follow the shape of the original ones, smoothing away some pointwise variability. The 95% credible band for one of them is also displayed. As we expected from the previous tests, the credible interval is very short, denoting a precise and reliable representation of the functional data.

Figure 4.25 reports a summary for $\boldsymbol{\beta}$s, $\boldsymbol{\mu}$ and τ_ε^2, all are characterized by short credible intervals, the ones for $\boldsymbol{\beta}$ parameters are implicitly shown in the credible bands of Figure 4.24. We do not report any traceplot because they are present no significant difference with respect

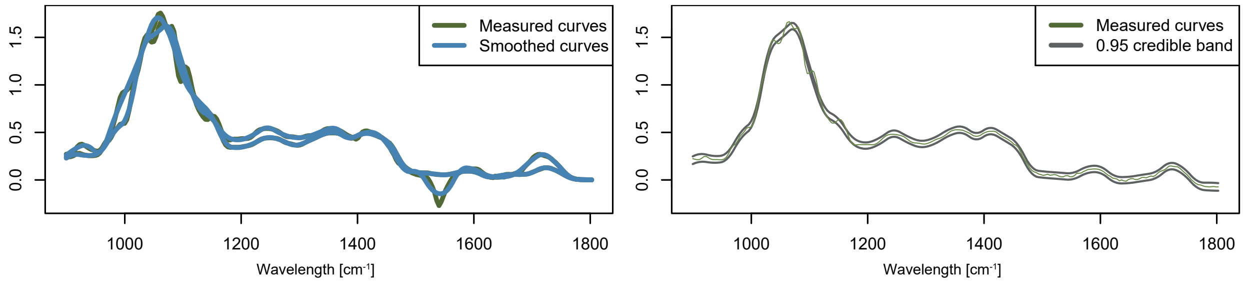

Figure 4.24. An example of two smoothed curves (left panel) obtained via `BGSL` Block DRJ. In the right panel we show the 95% credible band for one of them.

to the ones displayed in Figure 4.15, all chains reached convergence and swipe well the high posterior region.

Moreover, for the graphical step there are no significant differences with respect to the final estimate obtained exploiting `BDgraph`. This is not very surprising because the functional form of those data is very similar to the examples previously described and in all those cases we achieved a very high quality smoothing with no significant change with respect to the type of chain we used to navigate the space of all possible graphs. Let us now analyze the

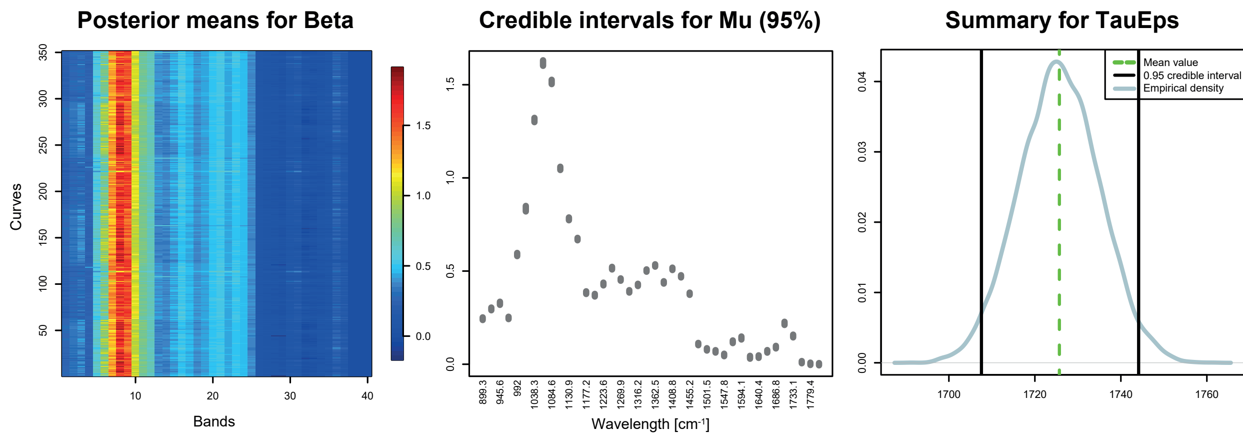

Figure 4.25. The leftmost plot shows the posterior means for all $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n$ coefficients, for each curve, the p values for the corresponding $\boldsymbol{\beta}_i$ are reported along the i-th row. The middle panel presents the 95% credible intervals for each component of $\boldsymbol{\mu}$, the rightmost one is a summary for τ_ε^2 containing its posterior mean, the 95% credible interval and the empirical density of the sampled values.

graphical related parameters, the graph G and the precision matrix $\boldsymbol{K}$. The latter is shown in Figure 4.26. As usual, we present two different plots, one contains only the off-diagonal elements, while the diagonal ones are instead reported in a separated panel, along with their credible intervals. As previously mentioned, the precision matrix obtained using Block DRJ is compared to the one obtained using `BDgraph` but, as for the regression parameters, also this time we do not evidence significant differences. The heat maps of the off-diagonal elements look the same, it is even difficult to see any difference with the naked eye.

The same reasoning holds for the diagonal elements. They all assess slightly above 300, except four of them that are related to the bands involved in the ascending and the central part of the main peak, which indeed is the part of the domain with major differences between the curves. Both solutions are sparse matrices, as expected, the one of `BDgraph` is able to shrink a little more, but the difference is not remarkable. It is visible only for the rightmost stripe, the one describing the long-term interactions with the last three bands, which is not perfectly squeezed to zero in the Block DRJ solution.
Such considerations are confirmed by a low value of the Kullback-Leibler divergence

$$\text{KL}\left(\hat{\boldsymbol{K}}_{BD},\ \hat{\boldsymbol{K}}_{BGSL}\right) = 0.00937$$

where $\hat{\boldsymbol{K}}_{BD}$ is the `BDgraph` estimate and $\hat{\boldsymbol{K}}_{BGSL}$ the one obtained from Block DRJ.

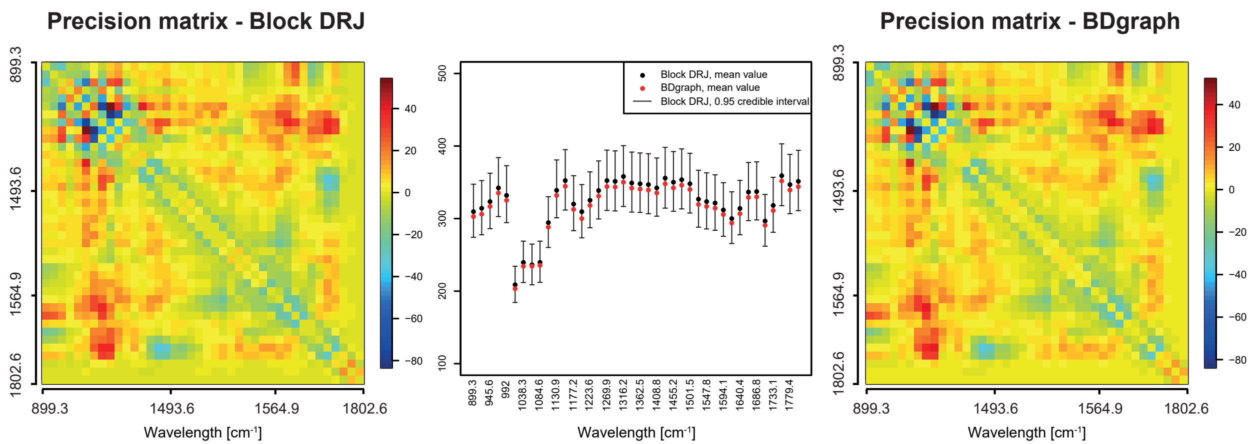

Figure 4.26. Estimates for the precision matrix $\boldsymbol{K}$ achieved through Block DRJ (left panel) and compared to the one of `BDgraph` (right panel). Both represent only the off-diagonal elements, the diagonal ones are reported in the middle panel. The black dots are the values obtained with Block DRJ and the grey lines are their 95% credible intervals. The red dots are the pointwise estimates obtained using `BDgraph`.

Figure 4.27 shows the posterior estimate of the block structured graph derived using our Block DRJ, the traceplot of the graph sizes visited after the burn-in period. As usual, we compare ourselves to the `BDgraph` solution, whose final estimate is given in the first panel of Figure 4.27.
The traceplot confirms that the chain has been runned long enough to reach convergence. The presence of some spikes is welcomed because it means that the chain is swiping well the high probability region and that, even though sometimes it takes a path of lower probability, it is able to leave it quickly. This property is not trivial due to the low acceptance rate that characterizes the chain, looking at the traceplot we see indeed that it is often flat. In this case, we registered an acceptance rate around 5%.
The structure of the two final estimates reflects what already observed in one of the experiments

of Section 4.2. Both achieved a sparse form that is also characterized by a block structure. Moreover, they seem to agree with each other: most of the blocks are located close to the diagonal, which was expected since close wavelength bands are likely to be associated with similar components and therefore to share interactions. We refer to such discoveries as short-term interactions. In particular, both graphs evidence a large amount of such interactions in the range $[1000 - 1200]\, cm^{-1}$ that is the one related to the main peak. The big size of this block suggests that the interactions between the corresponding bands are not only due to the overlap of spline supports, rather that the absorbance effects of several species overlay. From the heat maps in Figure 4.26 we see that those interactions are not trivial since both positive and negative dependencies are involved. They are not negligible, the highest precision values are indeed attained in this region. We also highlight a couple of blocks along the final part of the diagonal.

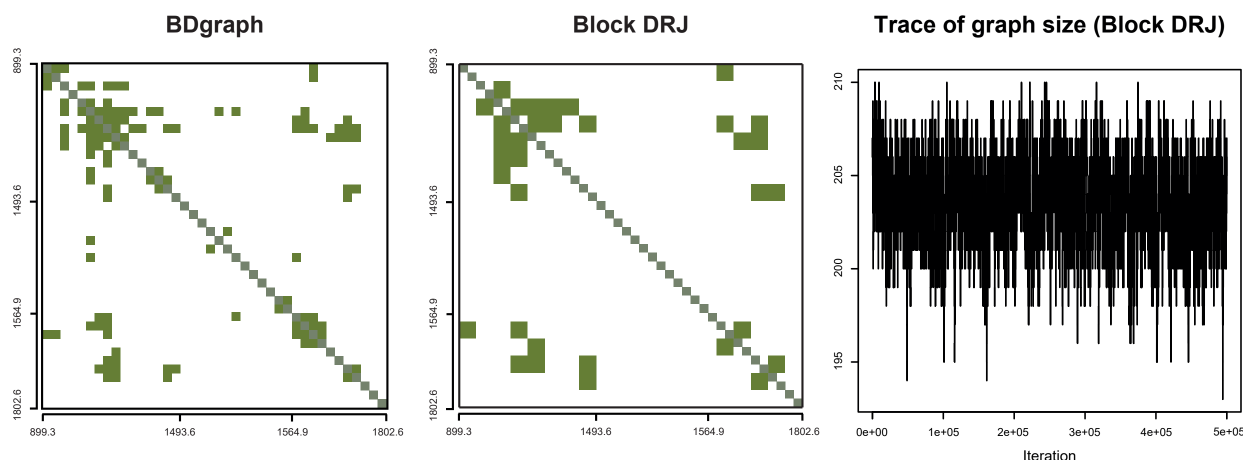

Figure 4.27. The middle panel presents the posterior estimate of the graph obtained with `FGMsampler` from `BGSL` using the Block DRJ procedure for the graphical step. The rightmost panels shows the corresponding traceplot. The leftmost panel contains the graph obtained exploiting `BDgraph` for the graphical step.

However, the peculiarity of a graphical model is the possibility of investigating long-term interactions and we indeed found out some off-diagonal blocks. The largest one corresponds to the interaction between the bands at the higher peak and the others at two lower peaks at around 1564.9 cm^{-1} and 1700 cm^{-1}, respectively. Indeed, with the infrared spectrometer, the same chemical species can exhibit peaks at different wavelengths and our analysis identifies which peaks should be analysed together. The one below is interesting for a twofold reason: it is the only identified long-term interaction that does not involve the main peak. It bounds the band at around 1500 cm^{-1}, which is where the curves suddenly decrease, and the final ones, where many curves show a final peak. Secondly, the heat map reveals that it is the only off-diagonal block denoted by negative values.

Finally, we mention that in the estimated graph there is a wide area with almost no interactions.

This is likely because the substances whose absorbance is in that range are not involved with the aforementioned peaks. Graphically, this is the central region where all data are rather flat and regular.
The last long-term block interaction identified is also the one between the farthest ones, it bounds together the very first band at 899.3 cm^{-1} and one of the latest, at 1700 cm^{-1}.

As in Section 4.2, see Figure 4.8, the two graphs share a similar block structure but the one of `BDgraph` is fragmented. They both agree about the presence of the large diagonal block, moreover the long-term interactions we mentioned above are displayed in both cases, with the clear difference that Block DRJ estimates complete blocks and `BDgraph` does not. This is one of those situations that mostly inspired our modelling choice of estimating only complete or empty graphs: we identified each portion of the domain, which in turn corresponds to the absorbance frequency of a specific chemical group, with a single $\boldsymbol{\beta}$ coefficient and we are investigating the relationship among them. Nevertheless, we know that the spline actually overlaps one another and that there may not be a perfect correspondence between each band we identified and a single substance, therefore it would be hard to explain the missing edge in what is clearly a block structure. Such interpretations are also conditioned by the fact that the experiments in Section 4.2 highlighted that `BDgraph` is inclined to false negativeness. The solution we propose tackles this problem by completing the most interesting blocks and removing the non-meaningful ones, where the degree of importance is naturally encoded and decided by the chain, it is not a posteriori decision. Looking at Figure 4.27 we also notice that some isolated links are filtered away. We may interpret the Block DRJ solution as a zoom out of the `BDgraph` one; it has a lower resolution and it is less detailed, but it allows to see such a structure from a wider prospective, enabling a simpler interpretation.

To assess the robustness of our solution, we run different instances of Block BGSL but changing the starting points of the chains and the prior on the graph. The example displayed in Figure 4.27 was obtained using a uniform, non-informative prior on the graph. A common choice in the literature (Wang et al. 2015; Cremaschi et al. 2019) is instead to use a Bernoulli prior, see (1.13), with parameter $\theta = 2/(p-1)$ to encourage a priori sparsity in the graph (Jones et al. 2005). We apply a similar choice but using a truncated Bernoulli prior, see Section 3.4.1, to ensure the block structure.
The additional scenarios we take into account are:

(a) The initial graph is not empty but it is chosen to be complete. The corresponding precision matrix is drawn from a Wishart distribution.

(b) We use a truncated Bernoulli prior for the graph, $\pi_{tb}(\theta)$, with $\theta = 2/(p-1)$.

(c) It combines **(a)** and **(b)**, the chain starts from a complete graph, the same graphical prior $\pi_{tb}(\theta)$ of the previous case is applied.

The estimated precision matrices are all compared to $\hat{\boldsymbol{K}}_{BD}$ using as usual the Kullback-Leibler divergence. The graphs are judged in terms of their Standardized-SHD with respect to the reference graph $\hat{G}_{BGSL}$ we obtained with Block DRJ, that is shown in Figure 4.27, middle panel. All such cases are summarized in Table 4.5. For what concerns the precision matrix,

	KL($\hat{\boldsymbol{K}}_{BD}$, $\hat{\boldsymbol{K}}$)	Std-SHD($\hat{G}_{BGSL}, \hat{G}$)
(a)	0.00947	0.067
(b)	0.01009	0.051
(c)	0.00912	0.041

Table 4.5. Three different chains are runned. (a) starts from a complete graph a precision randomly chosen according to a `Wishart` distribution. (b) uses a truncated Bernoulli prior with parameter $\theta = 2/(p-1)$. (c) combines those two aspects. $\hat{\boldsymbol{K}}$ and $\hat{G}$ are the final estimates for precision matrix and graph, respectively.

the differences are so small that we can conclude that all chains converge to the same matrix. The graph estimation does not always provide the same graph, some instabilities are present. However, the number of different links is not particularly significant and it is in line with the variability we registered in Section 4.2. Moreover, they all agree about the general structure, some variations have been registered for the size of the off-diagonal blocks and for the edges to be included in the final part of the diagonal.

To conclude, we also repeated the analysis using a smaller number of bases to show different granularity levels of our solution. Indeed, Figure 4.28 confirms that the main structure of the solution is always characterized by the presence of a large block on the diagonal and an additional extra-diagonal one. As the support of the basis elements becomes smaller, the estimated graph structure refines.

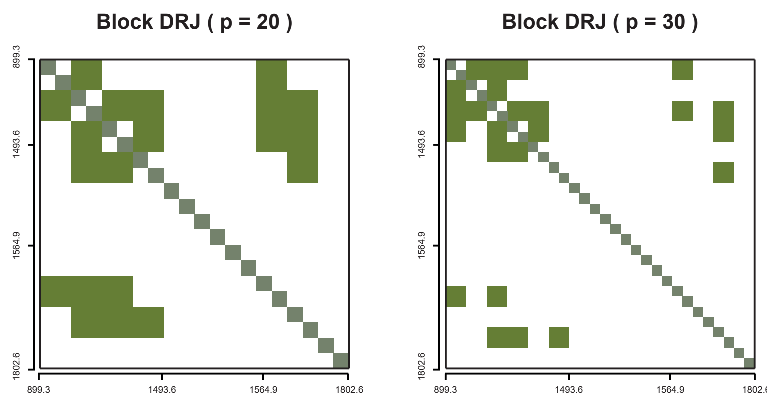

Figure 4.28. Estimated graphs using smaller graphs. In particular, we one on the lest is generated using $p = 20$ basis functions, the one on the right with $p = 30$.

Chapter 5

Discussion

We introduced some novel aspects in the framework of structural learning with Gaussian graphical models. Our main contribution has been the proposal of a new approach that allows to estimate graphs that have a particular block structure, leading to estimates that are more simple to be interpreted. Thanks to its dimensionality reduction of the graph space, this kind of graphical search is usually able to get sharper solutions, in particular with respect to the F_1-score index.

From a mathematical and modelling point of view, our **Block Reversible Jumps for GGM** is a trans-dimensional Monte Carlo Markov chain on the joint space of graphs and precision matrices that at each iteration modifies not just a single link but an arbitrary number of them. This feature is totally new to the literature, all existing methods we are aware of are only able to change one link at a time.

One side effect of our modeling assumptions is that we are no longer able to identify the precise structure within each block, the groups can only be fully connected or not connected at all. This kind of solution may look a bit coarse, but as soon as the dimension of the graph grows, it helps to understand the global behaviour of the phenomenon generating the data, rather than looking for all possible, even meaningless, relationships.

A drawback of the current formulation of the model is for sure the low acceptance rate it suffers from, which entails that the corresponding chains have to be runned for many iterations, implying non-negligible additional computational costs. This problem gets much worse as the size of the groups increases, in practice, at the moment the number of groups is limited to be only $p/2$ or $p/3$. This issue is strictly related to another weak spot of our proposal distribution, that is the need of tuning the σ_g^2 parameter, which defines the perturbation of the elements of the precision matrix that are modified in the construction of the proposed state. This parameter plays a key role in the definition of the chain, but, unfortunately, its value has to be fixed a priori. A further limitation is that all elements are perturbed according to the same σ_g^2, instead it would be desirable to use a specific value for each link, but then it

would then be impossible to tune all of them. An alternative approach may be instead to use an adaptive scheme.

The second major novelty introduced in this work is that we also successfully placed the graphical model over unobserved variables rather than the observed data as it is usually done. Within the setting of functional data, and in particular in the smoothing procedure using a B-spline basis expansion, we indeed modelled the regression coefficients by means of a GGM, obtaining a twofold advantage. First of all, thanks to the compact support of the basis functions, we are now able to investigate hidden inter-dependencies among different bands of the curve and to represent it through a block structured graph. Secondly, our hierarchical formulation of the smoothing problem allows us to borrow strength across curves taking into account also non-local interactions, a feature that may be useful, for example, in situations when dealing with sparse data and measured on different grids. A classical smoothing procedure usually relies only on local interactions that sometimes are not enough to fully explain the underlying phenomenon.
A graphical description of functional data is not new to the literature, but most of the current works use a graph to describe the relationship among functional objects. For example, Qiao et al. (2019); Zhu et al. (2016) used a graph to describe the connections among different areas of the brain in alcoholic and nonalcoholic subjects. In those cases, every node describes a particular location of the brain that is associated with a functional data, measured multiple times. This very sophisticated model is used to describe data, which is not our intent as we used it to explain the structure of unobserved variables, which makes our strategy innovative.

In this work, we only focused on a scenario such that the groups are fixed a priori, thanks to previous knowledge about the problem. It is for sure a limiting assumption that reduces the applicability of our method. A first avenue of research is to consider the possibility of a random partition of the variables into groups. A promising way to accomplish this extension is to model the graph as a Stochastic Block Model. This is a well-established model in network analysis that partitions the nodes in mutually exclusive groups such that the off-diagonal elements of the adjacency matrix are modelled as independent Bernoulli, whose probability depends only on the group membership of the involved vertices. The natural procedure to define the partition is to consider a Dirichlet-Multinomial distribution or some nonparametric extension of it. Our idea is instead to describe the groups as a changepoint process; the length of each group can be modelled through a geometric distribution, which induces a random partition of the nodes. This choice adds a new level in the hierarchical model, the first step of the new sampling strategy would then be the update of those groups, which could be done using an adaptive sampler, as the one described in Benson et al. (2018). Once this step is completed, graph and precision matrix can be updated using one of the methods studied in this work.
A second possible improvement to the current technique would be to reformulate the Block

DRJ within the setting of a continuous time Birth and Death chain. In that way, it could not suffer from low acceptance rates because it is not an acceptance-rejection sampler. This would pave the way to a more efficient shrinking estimator for the precision matrix and the posterior probabilities of edge inclusion, as discussed in section 3.5.4. The task is not trivial because it would require to deal with the complex theory of spatial jump processes by Preston (1977) that we briefly presented in Section 1.3.3. Finally, a further extension can consider other types of graphical models, such as log-linear models or to non-Gaussian data by using a copula transition. Such features are already available in the package `BDgraph`. It would be interesting to extend them in the framework of block structured graphs.

All implemented methods are collected in our `R` package called `BGSL`. Its purpose is to provide the user different Monte Carlo Markov chain samplers for dealing with Gaussian graphical models in many different settings: the classical procedures using non-structured graphs, as well as our new proposal, the block structured case and the functional graphical models. They both are totally new to the literature, implying that `BGSL` is a rather unique package in the field of Gaussian graphical models. It also means that it is far from being an exhaustive analysis of the problem, several embellishments are needed also from a computational point of view.
We wrote all `BGSL` code from scratch and it has been developed taking into account that this project would require to be extended and improved. This is achieved by exploiting as much as possible template programming and static or dynamic polymorphic classes. In this way, its implementation is composed of separate modules which can be independently modified without compromising the operations of the others. Indeed, thanks to the code generality, the changes are automatically integrated in the remaining ones. Right now, this feature is not already available for every possible class, many improvements can still be done in that direction. A common problem in Bayesian statistics and in particular with Gaussian graphical models is the way results are stored. `BGSL` deals with this problem by writing all sampled values to a binary HDF file. It is a possible solution, it may not be the best one and for sure we were not able to really exploit all functionalities of the **HDF5** library.
Despite the possible limitations of the model proposed and discussed in this dissertation, we believe that our proposal constitutes a significant improvement to the literature of Gaussian graphical models, providing a solid starting point for researchers and developers in the field.

Acronyms

GGM	Gaussian graphical models
BGSL	Block Graph Structural Learning
MCMC	Monte Carlo Markov chain
SSS	Shotgun Stochastic Search
MOSS	Mode Oriented Stochastic Search
BDMCMC	Birth and Death Monte Carlo Markov chain
MH	Metropolis-Hastings
ARMH	Add-Remove Metropolis Hastings
Block ARMH	Block Add-Remove Metropolis Hastings
RJMCMC	Reversible Jumps Monte Carlo Markov chain
RJ	Reversible Jumps (for Gaussian graphical models)
Block RJ	Block Reversible Jumps (for Gaussian graphical models)
DRJ	Double Reversible Jumps (for Gaussian graphical models)
Block DRJ	Block Double Reversible Jumps (for Gaussian graphical models)
CGL	Cluster Graphical Lasso
FGM	Functional Graphical Model
FLM	Functional Linear Model
GSL	GNU Scientific Library
HDF5	Hierarchical Data Format, version 5

Bibliography

Atay-Kayis A, Massam H (2005) A Monte Carlo method for computing the marginal likelihood in nondecomposable Gaussian graphical models. Biometrika 92:317–335

Baldi P, Brunak S, Chauvin Y, Andersen CA, Nielsen H (2000) Assessing the accuracy of prediction algorithms for classification: an overview. Bioinformatics 16(5):412–424

Barbieri MM, Berger JO (2004) Optimal predictive model selection. Annals of Statistics 32(3):870–897

Benson A, Friel N, et al. (2018) Adaptive mcmc for multiple changepoint analysis with applications to large datasets. Electronic Journal of Statistics 12(2):3365–3396

Bhadra A, Mallick BK (2013) Joint high-dimensional Bayesian variable and covariance selection with an application to eQTL analysis. Biometrics 69(2):447–457

Bornn L, Caron F, et al. (2011) Bayesian clustering in decomposable graphs. Bayesian Analysis 6(4):829–846

Burns P (2011) The R inferno. Lulu. com

Cappé O, Robert CP, Rydén T (2003) Reversible jump, birth-and-death and more general continuous time Markov chain Monte Carlo samplers. Journal of the Royal Statistical Society: Series B (Statistical Methodology) 65(3):679–700

Chung FR, Graham FC (1997) Spectral graph theory. 92, American Mathematical Soc.

Codazzi L, Colombi A, Gianella M, Argiento R, Paci L, Pini A (2021) Functional graphical model for spectrometric data analysis. arXiv preprint arXiv:210311666

Cremaschi A, Argiento R, Shoemaker K, Peterson C, Vannucci M (2019) Hierarchical normalized completely random measures for robust graphical modeling. Bayesian Analysis 14(4):1271–1301

Dellaportas P, Giudici P, Roberts G (2003) Bayesian inference for nondecomposable graphical gaussian models. Sankhyā: The Indian Journal of Statistics pp 43–55

Devijver E, Gallopin M (2018) Block-diagonal covariance selection for high-dimensional gaussian graphical models. Journal of the American Statistical Association 113(521):306–314

Dobra A, Hans C, Jones B, Nevins JR, Yao G, West M (2004) Sparse graphical models for exploring gene expression data. Journal of Multivariate Analysis 90(1):196–212

Dobra A, Lenkoski A, Rodriguez A (2011) Bayesian inference for general gaussian graphical models with application to multivariate lattice data. Journal of the American Statistical Association 106(496):1418–1433

Eddelbuettel D, François R (2011) Rcpp: Seamless R and C++ integration. Journal of Statistical Software 40(8):1–18, DOI 10.18637/jss.v040.i08, URL http://www.jstatsoft.org/v40/i08/

Friedman J, Hastie T, Tibshirani R (2008) Sparse inverse covariance estimation with the graphical lasso. Biostatistics 9(3):432–441

Giudici P, Castelo R (2003) Improving Markov chain Monte Carlo model search for data mining. Machine learning 50(1-2):127–158

Green PJ (1995) Reversible jump markov chain monte carlo computation and bayesian model determination. Biometrika 82(4):711–732

den Hertog D, Roos C, Terlaky T (1993) The linear complimentarity problem, sufficient matrices, and the criss-cross method. Linear Algebra and Its Applications 187:1–14

Holland JK, Kemsley EK, Wilson RH (1998) Use of Fourier transform infrared spectroscopy and partial least squares regression for the detection of adulteration of strawberry purées. Journal of the Science of Food and Agriculture 76(2):263–269

Jones B, Carvalho C, Dobra A, Hans C, Carter C, West M (2005) Experiments in stochastic computation for high-dimensional graphical models. Statistical Science 20:388–400

Kullback S, Leibler RA (1951) On information and sufficiency. The annals of mathematical statistics 22(1):79–86

Kumar S, Ying J, de Miranda Cardoso JV, Palomar DP (2020) A unified framework for structured graph learning via spectral constraints. Journal of Machine Learning Research 21(22):1–60

Kusnierczyk W (2012) rbenchmark: Benchmarking routine for R. URL https://CRAN.R-project.org/package=rbenchmark, r package version 1.0.0

Lauritzen SL (1996) Graphical models. Oxford University Press, Oxford

Lenkoski A (2013) A direct sampler for G-Wishart variates. Stat 2(1):119–128

Lenkoski A, Dobra A (2011) Computational aspects related to inference in Gaussian graphical models with the G-Wishart prior. Journal of Computational and Graphical Statistics 20(1):140–157

Letac G, Massam H, Mohammadi R (2017) The ratio of normalizing constants for Bayesian graphical Gaussian model selection. preprint arXiv:170604416v2

Liu F, Chakraborty S, Li F, Liu Y, Lozano AC, et al. (2014) Bayesian regularization via graph laplacian. Bayesian Analysis 9(2):449–474

Madigan D, York J, Allard D (1995) Bayesian graphical models for discrete data. International Statistical Review/Revue Internationale de Statistique pp 215–232

Mohammadi A, Wit EC (2012) Gaussian graphical model determination based on birth-death mcmc inference. arXiv preprint arXiv:12105371

Mohammadi A, Wit EC (2015) Bayesian structure learning in sparse Gaussian graphical models. Bayesian Analysis 10(1):109–138

Mohammadi R, Wit EC (2019) BDgraph: An R package for Bayesian structure learning in graphical models. Journal of Statistical Software 89(3):1–30, DOI 10.18637/jss.v089.i03

Møller J, Pettitt AN, Reeves R, Berthelsen KK (2006) An efficient markov chain monte carlo method for distributions with intractable normalising constants. Biometrika 93(2):451–458

Müller P, Parmigiani G, Rice K (2007) FDR and Bayesian multiple comparisons rules. In: Bernardo JM, Bayarri M, Berger J, Dawid A, Heckerman D, Smith A, West M (eds) Bayesian Statistics 8, Oxford University Press, Oxford

Murray I, Ghahramani Z, MacKay D (2012) Mcmc for doubly-intractable distributions. arXiv preprint arXiv:12066848

Ni Y, Stingo FC, Baladandayuthapani V (2019) Bayesian graphical regression. Journal of the American Statistical Association 114(525):184–197

Peterson C, Stingo FC, Vannucci M (2015) Bayesian inference of multiple Gaussian graphical models. Journal of the American Statistical Association 110(509):159–174

Piccioni M (2000) Independence structure of natural conjugate densities to exponential families and the gibbs' sampler. Scandinavian journal of statistics 27(1):111–127

Powers DM (2020) Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. arXiv preprint arXiv:201016061

Preston C (1977) Spatial birth and death processes. Bulletin of the International Statistical Institute 46:371–391

Qiao X, Guo S, James GM (2019) Functional graphical models. Journal of the American Statistical Association 114(525):211–222

R Core Team (2013) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL http://www.R-project.org/

Ramsay J, Silverman B (2005) Functional Data Analysis, Second Edition. Springer, New York

Roverato A (2002) Hyper inverse Wishart distribution for non-decomposable graphs and its application to Bayesian inference for Gaussian graphical models. Scandinavian Journal of Statistics 29(3):391 – 411

Scott J, Carvalho C (2008) Feature-inclusion stochastic search for Gaussian graphical models. Journal of Computational and Graphical Statistics 17(4):790–808

Sisson SA, Fan Y (2011) Reversible jump mcmc. In: Handbook of Markov Chain Monte Carlo, Chapman and Hall/CRC, pp 89–113

Tan KM, Witten D, Shojaie A (2015) The cluster graphical lasso for improved estimation of gaussian graphical models. Computational statistics & data analysis 85:23–36

Tsamardinos I, Brown LE, Aliferis CF, Moore AW (2006) The max-min hill-climbing Bayesian network structure learning algorithm. Machine Learning 65(1):31–78

Uhler C, Lenkoski A, Richards D (2018) Exact formulas for the normalizing constants of wishart distributions for graphical models. The Annals of Statistics 46(1):90–118

Wang H (2010) Sparse seemingly unrelated regression modelling: Applications in finance and econometrics. Computational Statistics & Data Analysis 54(11):2866–2877

Wang H, Li SZ (2012) Efficient Gaussian graphical model determination under G-Wishart prior distributions. Electronic Journal of Statistics 6:168–198

Wang H, et al. (2015) Scaling it up: Stochastic search structure learning in graphical models. Bayesian Analysis 10(2):351–377

Wickham H, Danenberg P, Csárdi G, Eugster M (2020) roxygen2: In-Line Documentation for R. URL https://CRAN.R-project.org/package=roxygen2, r package version 7.1.1

Wild B, Eichler M, Friederich HC, Hartmann M, Zipfel S, Herzog W (2010) A graphical vector autoregressive modelling approach to the analysis of electronic diary data. BMC medical research methodology 10(1):28

Xia Y, Cai T, Cai TT (2018) Multiple testing of submatrices of a precision matrix with applications to identification of between pathway interactions. Journal of the American Statistical Association 113(521):328–339

Yang J, Zhu H, Choi T, Cox DD (2014) Smoothing and mean-covariance estimation of functional data with a bayesian hierarchical model. arXiv preprint arXiv:14025723

Yang J, Cox DD, Lee JS, Ren P, Choi T (2017) Efficient bayesian hierarchical functional data analysis with basis function approximations using gaussian–wishart processes. Biometrics 73(4):1082–1091

Yuan M, Lin Y (2007) Model selection and estimation in the gaussian graphical model. Biometrika 94(1):19–35

Zheng W, Shu H, Tang H, Zhang H (2019) Spectra data classification with kernel extreme learning machine. Chemometrics and Intelligent Laboratory Systems 192:103815

Zhu H, Strawn N, Dunson DB (2016) Bayesian graphical models for multivariate functional data. The Journal of Machine Learning Research 17(1):7157–7183

Ringraziamenti

Ringrazio le professoresse Guglielmi, Paci e Pini e il professor Argiento per la disponibilità, per i puntuali consigli tecnici e per aver coltivato la mia passione verso la materia, dandomi la fiducia necessaria per fare scelte che non avrei avuto il coraggio di fare e creando un ambiente di collaborazione in cui ho potuto esprimermi al meglio.

Ringrazio i miei colleghi ed amici Laura e Matteo, le cui ricerche costituiscono la base di partenza di questo elaborato. Inoltre, sono molto riconoscente per l'aiuto fornitomi da Matteo nello sviluppo del pacchetto `R`.

Si conclude un ciclo di studi quanto mai intenso, le cui difficoltà sono andate anche oltre l'ambito accademico ma che per fortuna non ho mai dovuto affrontare da solo. Per questo ringrazio la mia famiglia, per aver supportato me e la mia educazione con tanto entusiamo e comprensione. Grazie per aver tracciato quel sentiero che mi ha portato ad essere una persona di cui andare fiero. Una grande fortuna che ho avuto è che il mio percorso formativo sia stato affidato a persone esemplari che mi hanno trasmesso il meglio di loro. Sono tanti i maestri, allenatori e professori che vorrei ringraziare, ma tra tutti penso che un ruolo di particolare rilevanza l'abbiano avuto le professoresse Pavesi e Giordano.

Ringrazio inoltre tutti i miei amici di ADA, tutti i miei vecchi compagni di classe e quelli nuovi di Ingegneria Matematica per aver sempre creduto in me, per essere un punto di riferimento, per aver impedito che la paura diventasse panico nonché per essere degli straordinari esempi di vita.

Un pensiero speciale va ai miei inseparabili compagni di squadra, Andrea, Giovanni, Matteo e Martino, ai miei amici più cari e insostituibili, Silvia e Federico e al grande aiuto ricevuto da parte di Teresa, Laura, Margherita, Martina, Chiara e Francesco.
E soprattutto a te Bea, il cui sostegno e amore non è mai venuto meno, ha solo cambiato forma.

Con affetto, Alessandro.