Scalable Bayesian Hierarchical modelling with application in genomics

Antonia Kontaratou

Thesis submitted for the degree of
Doctor of Philosophy

School of Mathematics, Statistics & Physics
Newcastle University
Newcastle upon Tyne
United Kingdom

November 2020

This thesis is dedicated to my family.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Darren Wilkinson, for his valuable help and guidance over the course of my postgraduate research. I would also like to express my thanks to the teaching and administrative staff of the Centre for Doctoral Training in Cloud computing for Big Data for their kindness and support, as well as my fellow PhD students for their friendship. In addition, I would like to thank Professor David Lydall and Dr Conor Lawless for their valuable comments on the biological content. I am grateful to the Engineering and Physical Sciences Research Council for funding my research [grant number EP/L015358/1].

All these would not have happened without the unwavering love and encouragement from my family and particularly my mother, whom I am lucky to have. Finally, I would like to thank my friends and my partner for their patience and support.

Abstract

Hierarchical modelling can be applied to data organised in groups, for which we are interested in describing the within and between group variability. This type of model is very useful for a broad range of statistical problems. However, due to the complex nature of some data and the continuously increasing volume of datasets, using current methodologies for Bayesian hierarchical modelling can be challenging. The algorithms currently utilised, such as the Markov Chain Monte Carlo (MCMC) family, can be computationally intensive and difficult to parallelise, often leading to extended processing times, limiting exploration of different models, especially in cases of "Big Data" applications. These algorithms can be deployed using various programming paradigms, such as object-oriented, probabilistic and functional. The latter has been gaining ground in academia and industry over recent years. This thesis is concerned with examining an approach that will harness the benefits of functional programming and aims to provide valuable insights on whether MCMC algorithms, and in particular the Gibbs sampler, implemented in a functional style, can scale better whilst remaining accurate. More specifically, we implement a Gibbs sampler in Scala to fit a Bayesian hierarchical two-way Anova model that includes interactions and accounts for various levels of asymmetry in the effects. We incorporate variable selection on the interaction effects through exploration of two techniques, an indicator variable approach, and the Horseshoe prior. In addition, we investigate under which model specifications parallelism can affect speed-up. After comparing the efficiency of the methods developed to the results deriving from some already existing libraries that automate and facilitate the modelling and inference processes, we explore their application on a yeast genome case study. The identification of gene complexes that genetically interact with telomere capping defects is of great importance in cell biology, as research has shown that telomeres can be related to ageing and various diseases. A Bayesian hierarchical model is developed to highlight and estimate the strength of potential epistatic relationships between genes of interest. However, the methodology developed has a wider range of applications and is not limited to the yeast genome case study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Vast amounts of data are being collected and shared in research, industry and society. The increasing volume of data and complexity of models, along with the evolution of technology and cloud resources, present new needs and opportunities to transform traditional statistical learning, modelling and prediction techniques.

Bayesian statistics is a field of statistics based on the idea that uncertainty is expressed through probability distributions that are updated in the light of new information. The Bayesian framework is widely used to gain insights about high-dimensional problems by holistically modelling the system. Among the models that are often encountered are hierarchical models, which are used when data are grouped in some way and where understanding their dependence and variability is important. Bayesian inference is highly dependent on integration and involves difficult integrals, the dimension of which increases along with the dimension of the problem, rendering them analytically intractable. Therefore, simulation techniques, such as the Markov Chain Monte Carlo (MCMC) family of algorithms, are used to numerically approximate the quantities of interest. Bayesian methods are often computationally and time demanding despite the advances in technological infrastructure. A lot of research has been done towards the development of strategies that can alleviate potential memory and processing constraints associated with Bayesian methods. Some scaling methodologies focus more on improving the algorithmic steps and reformulating the mathematical background of these methods. Other approaches emphasise using modern technological resources to distribute the computational workload and render Bayesian simulation techniques more efficient.

Scalability is a term broadly used with an interpretation that slightly varies depending on the field and context. In the regime of combining statistical modelling with computing science, scalability can be expressed as the ability of a system to adapt to more complex models and increased data size and dynamically take advantage of the available technology to efficiently capture the structure of data. Developing and implementing scalable and multi-processor algorithms is a challenge since it requires knowledge of both the problem domain and the development tools. Well-designed programs can achieve scalability irrespective of the choice of programming language. However, some programming paradigms and language-specific features can render scalability an easier and more efficient task than others. One example is functional programming, which has become more popular in recent years. The functional approach suggests that programs should be built by composing smaller functions that do not use or modify shared global states and do not have side-effects. Using pure functions of this type facilitates the design and implementation of large-scale projects by ensuring that scalability will be less prone to errors.

The main objective of this research project is to explore whether and how Bayesian inference used for hierarchical models can become more scalable with the use of functional programming and more specifically the Scala programming language. In particular we focus on the Gibbs sampler, a Markov Chain Monte Carlo algorithm commonly used for Bayesian statistical inference. The main model explored is a variation of a classic two-way Anova. It involves two categorical explanatory variables with various levels, and their corresponding interactions on which we apply variable selection using two Bayesian techniques, variable selection indicators and adaptive shrinkage priors. In addition, we account for different levels of symmetry in the effects, according to the rationale behind the experiment of our case study which is related to yeast genomics. Bayesian hierarchical modelling has a wide range of applications, and efficient sampling processes can facilitate faster exploration of alternative models and statistical learning.

The remainder of this thesis is structured as follows. Chapter 2 presents some fundamental terms and concepts used in Bayesian statistics. After explaining Bayes' rule, which is the core of the Bayesian framework, we show how integration can lead to analytically intractable calculations. We then present the Markov Chain Monte Carlo class of algorithms that overcome this problem using repetitive sampling to approximate the distributions of interest. Additionally, we introduce a specific type of models, the hierarchical models on which we focus in this thesis. The chapter ends with presentation of variable selection methods and model comparison techniques used in the Bayesian paradigm that will be applied in later chapters.

Chapter 3 aims to introduce the two-way Anova models that tend to be quite complex when they include interactions of the multi-level categorical variables involved. Bayesian variable selection on the interaction effects can be then incorporated in the models to impose dimensionality reduction using a Gibbs sampler, a particular MCMC algorithm that draws samples from the full conditional distributions of the unknown parameters to approximate their posterior distributions. Two alternative variable selection techniques are explored, variable selection using indicators and in particular the Kuo and Mallick approach, and the Horseshoe prior that belongs in the category of adaptive shrinkage priors. Both models additionally account for four variations of symmetry in the effects, main and/or interaction effects being symmetric/asymmetric. We show how the full conditionals are calculated for these models, and build the statistical and mathematical background in order to develop, from the ground up, a Gibbs sampler in Scala under the specific modelling assumptions considered.

Chapter 4 provides a literature review to show why scaling Bayesian inference methods can be challenging. It presents the main tendencies among the scaling strategies that aim to make MCMC techniques more efficient and can be categorised to subsampling and divide-and-conquer techniques. A discussion of some methods that fall into each category aims to show commonalities and limitations.

Chapter 5 is devoted to functional programming in Scala, since this is the programming paradigm and language in which we chose to explore the deployment of Bayesian techniques. After highlighting some important aspects of this programming approach, that were taken into consideration during the development of the relevant code, we present some principal ideas of category theory that are embedded in some functional programming features to allow code re-usability. In the last section of this chapter, we see how probabilistic programming used for inference problems can be linked to functional programming.

In order to be able to evaluate the results and performance of the Gibbs sampler developed in Scala we implemented a Gibbs sampler for the same models in the long-established probabilistic programming language (PPL) JAGS (Just Another Gibbs Sampler) using R. Initially, variable selection was implemented using the indicators method along with the Gibbs sampler in both languages. Further exploration of the Horseshoe prior technique led to the development of the equivalent version for the two implementations. However, the Horseshoe prior approach is mainly intended to be used with MCMC algorithms that, due to their nature, do not support discrete variables and therefore cannot implement variable selection with discrete indicators, such as Hamiltonian Monte Carlo (HMC). Consequently, two additional implementations were explored using the R library of another well-known PPL, Stan, and Rainier, a probabilistic library in Scala, that both use HMC. Chapter 6 presents the implementational details of Bayesian hierarchical modelling for the alternative tools mentioned above, with emphasis placed on the Gibbs sampler developed in Scala, also referred to as TWIiS (Two-way Interactions in Scala). The latter constitutes a part of the project that a significant amount of time and work was devoted to, and for which we further developed a parallel version.

Chapter 7 presents the outcome of a simulation study used to illustrate, through examples with synthetic datasets, the application and results of the methodologies considered, as well as provide efficiency comparisons among them. The main focus is placed on JAGS and TWIiS. However, smaller examples are also considered for Stan and Rainier. After presenting how the simulation process was designed and the posterior analysis steps we followed, we firstly explore the models that use variable selection indicators and then we examine the alternative models with the Horseshoe priors. For each approach we explore three examples with varying complexity and data size to test their impact on the results and efficiency. An integrated comparison between JAGS and TWIiS for both variable selection techniques is presented in the last section of this chapter that aims to identify the most suitable and efficient approach for the models of interest.

Since the case study explored in this project is relevant to yeast genomic research, Chapter 8 is dedicated to setting the biological background and explaining the nature of the experiment that the main dataset derives from. MiniQFA (Quantitative Fitness Analysis) is an experiment conducted by the Institute for Cell and Molecular Biosciences of Newcastle University and is relevant to telomere biology. Telomeres are DNA sequences located at the ends of the chromosomes that protect the molecule during cell division. Telomere shortening and defects are part both of natural degradation and mutations. In miniQFA, scientists apply certain temperature sensitive mutations to yeast strains along with pair-wise gene deletions. Then, they record the colonies' growth and calculate their fitness. The aim is to use the available data and explore genetic interactions between the two deleted genes and particular mutations. Bayesian hierarchical modelling and variable selection can be used to highlight genetic interactions.

Chapter 9 presents the results of using TWIiS for fitting Bayesian hierarchical models to the miniQFA dataset. Various models with certain specifications and experimental conditions are explored and result in identification of pair-wise gene deletions that appear to interact with the telomere capping defect. The results are summarised through genetic interaction plots that illustrate which interactions cause phenotypic enhancement, suppression or have no impact on the biological phenomenon examined.

Finally, Chapter 10 considers the contribution of this thesis, summarises our main conclusions and suggests topics for future work.

Chapter 2

Bayesian inference and computation

Statistical inference is the process of using data, usually a sample of a population, in order to understand the underlying process and parameters that generated them, and consequently, deduce the characteristics of the population. Bayesian statistics, named after Thomas Bayes, is the field of statistics established on the principle that our uncertainty about an event occurring is expressed by probability, that can be updated under the light of new information.

In the following sections we are going to define some fundamental terms and concepts used in Bayesian inference. We will present the role of the Markov Chain Monte Carlo family of algorithms, and show why they are important in the Bayesian framework. Furthermore, we will introduce the concept of hierarchical models, Bayesian variable selection, and model comparison.

Sections 2.1 and 2.2, unless declared otherwise, are based on the following four comprehensive textbooks. *Bayesian Data analysis*, by Gelman *et al.* (2013), *A First Course in Bayesian Statistical Methods*, by Hoff (2009) and *A Student's Guide to Bayesian Statistics*, by Lambert (2018) that present in-depth the Bayesian framework, and *Markov chain Monte Carlo: stochastic simulation for Bayesian inference* by Gamerman and Lopes (2006) that provides a detailed guide to Markov Chain Monte Carlo methods for Bayesian statistics.

2.1 Key points of Bayesian inference

The fundamental formula of Bayesian logic is Bayes' rule. In order to build the theoretical background of Bayesian statistics, we will see how integration, which is involved in the theorem, can complicate calculations. In addition, we will explore ways to tackle this difficulty and facilitate the process of Bayesian learning.

2.1.1 Introductory terms in Bayesian statistics

In probability theory, the aim is to describe the behaviour of *random variables.* In the Bayesian framework all unknown quantities of interest are treated as random variables and uncertainty is expressed using probability. With every value of a random variable there is an associated numerical value, defining the probability of this event occurring. The functions that describe how these probabilities are distributed over all possible outcomes are called *probability distributions.* Discrete distributions encode probabilities by a *probability mass function (pmf)*, which gives the probability that the variable is exactly equal to some value. For continuous random variables the probability distribution is associated with a *probability density function (pdf)* which represents the probability that a variable lies within an interval of possible values, since the probability of a random variable being exactly equal to some value in the continuous space is zero. There are two conditions that have to be satisfied for a probability distribution to be proper:

- The values of the distribution must be real and positive.
- The overall probability must be 1. Hence, for discrete random variables, the sum of all probabilities must be 1. For continuous random variables, the integral of the probability density function must be 1.

In order to understand Bayesian inference and how its core theorem derives, it is important to clarify some rules of probability. Given two events A and B, the probability of both events occurring is denoted as $p(A \cap B)$ or $p(A, B)$ and is called *joint probability.* The probability of event A ($p(A)$) occurring irrespective of the outcome of B is the *marginal probability* of A. In addition, the probability of event A occurring given that B has already happened is called the *conditional probability* of A given B and is quantified as:

$$p(A|B) = \frac{p(A, B)}{p(B)}. \tag{2.1}$$

The joint probability is commutative so $p(A, B) = p(B, A)$, and rearranging the terms of 2.1 results in:

$$p(A|B) \times p(B) = p(A, B) = p(B, A) = p(B|A) \times p(A). \tag{2.2}$$

So,

$$p(A|B) = \frac{p(B|A) \times p(A)}{p(B)}. \tag{2.3}$$

The last equation (2.3) is Bayes' theorem, which is the principal formula used in Bayesian statistics. In the following section we will reintroduce Bayes' rule in the inference framework.

2.1.2 Bayes' rule and an example of Bayesian modelling

In Bayesian learning, similarly to general statistical induction, the aim is to learn about the characteristics of a population from a subset of that population, the sample. Out of all possible datasets Y, that are numerical descriptions of the subsets of that population, we obtain a single dataset y, and out of all possible parameter values from the parameter space Θ, we want to identify the parameter/parameters ($\theta/\boldsymbol{\theta}$) that best describe the true population characteristics.

2.1.2.1 Bayes' rule

The core idea of Bayesian statistics is the use of probability distributions to indicate our uncertainty about the unknown parameters and the underlying model. Therefore, our beliefs about y and θ are expressed through probability distributions. Models are characterised by their set of parameters, variations of which can change the system behaviour. Before the data are observed our beliefs about the unknown parameter θ are expressed through a parameter prior distribution $p(\theta)$, and after the data y are observed a parameter posterior distribution is specified as $p(\theta|y)$.

Formula 2.3 was not only the definition of the conditional probability of A given B, but also, a first definition of Bayes' rule. Incorporating the unknown parameter θ in place of A and conditioning on the data y results in the classical form of Bayes' rule used for inference as shown in 2.4.

$$\overbrace{p(\theta|y)}^{\substack{\text{Posterior}\\\text{dist.}}} = \frac{\overbrace{p(y|\theta)}^{\textit{Likelihood}} \times \overbrace{p(\theta)}^{\substack{\text{Prior}\\\text{dist.}}}}{\underbrace{p(y)}_{\substack{\text{Normalising}\\\text{factor}}}}. \tag{2.4}$$

In the Bayesian paradigm, as we can also see in 2.4, there are three main levels of uncertainty, in the form of the prior distribution, the likelihood/sampling model and the posterior distribution. So, the building blocks of Bayes' rule, considering a simple case with one unknown variable θ, are:

- The *prior distribution* $p(\theta)$ describes our belief that θ is the true value of the population characteristics that we want to estimate. It represents our uncertainty about the true parameter value before observing the data. Depending on our knowledge about the process we try to model, the prior distributions assigned can be uninformative, weakly informative or informative. The choice of prior distributions determines to a degree the extent of the impact of our pre-existing beliefs, and of the data sample itself, to the posterior distribution.

- The *likelihood* is a function of the parameter θ with the data y being fixed. There is a double notation for it, with the two terms, $\mathcal{L}(\theta|y) = p(y|\theta)$, being often used interchangeably. The left side is the likelihood of θ for a particular data sample. The right side, also known as sampling density, is the probability of this data occurring for any value of θ, since θ varies. Hence, the likelihood describes our belief that y would be the outcome if we knew that θ was the true value. The likelihood is not a proper probability distribution, it does not sum (for discrete) or integrate (for continuous) to one. To determine the likelihood for a particular problem a known distribution that best satisfies the assumptions about the data generating process is selected.

- The *posterior distribution* $p(\theta|y)$ describes our belief that θ is the true value after observing the data. It represents the fact that our beliefs about θ are updated under the light of new information. The posterior distribution is perceived as a weighted average of the prior and the likelihood. Often, for interpretational reasons, posterior distributions are described by their summary estimates or credible intervals. Furthermore, when new information is available, posterior distributions can be used as priors for a new Bayesian analysis.

- The *normalising factor* at the denominator is the term that ensures that the posterior probability is actually a proper distribution. The numerator's sum or integral over all parameter values is not necessarily equal to one. So, the denominator is the value of this sum or integral and normalises the numerator. If we denote the posterior space with Θ, there are two cases for $p(\theta)$:

$$p(y) = \begin{cases} \sum\limits_{\Theta} p(y,\theta) = \sum\limits_{\Theta} p(y|\theta) \times p(\theta), & \text{if } \theta \text{ is discrete} \\ \\ \int\limits_{\Theta} p(y,\theta) d\theta = \int\limits_{\Theta} p(y|\theta) \times p(\theta) d\theta, & \text{if } \theta \text{ is continuous.} \end{cases} \tag{2.5}$$

  Another way to consider the denominator is as a marginal distribution of the joint density of data and θ, where we have marginalised out all the dependence on the unknown parameter. Then, $p(y)$ represents a probability distribution for the possible data samples, given a model choice, before we observe a specific sample.

The posterior distribution returned from Bayes' rule (2.4), contains information about the whole geometry of the distribution. Sometimes, we might only be interested in the shape of the posterior, for example if we want to know the relative posterior density of two points. Then, Bayes' rule can be written in terms of proportionality to the numerator, which defines the shape, since the denominator does not depend on θ. This form of posterior is called *un-normalised* and is defined as:

$$\begin{gathered} p(\theta|y) \propto p(y|\theta) \times p(\theta), \\ \text{Posterior} \propto \text{Likelihood} \times \text{Prior}. \end{gathered} \tag{2.6}$$

2.1.2.2 A simple example of Bayesian modelling

As a simple example of the Bayesian paradigm we consider an unfair coin toss experiment, under the initial assumption that there is a probability density over the true probability of heads with mean $\mu = 0.7$ and variance $\sigma^2 = 0.1$. Suppose that we are interested in the probability of heads, denoted as θ. In order to use Bayes' rule we apply the following steps:

Step 1: Define the prior distributions of the unknown parameter.

First, we define the prior distribution by which we express our beliefs about the unknown parameter based on previous experience. We use a Beta distribution which is defined in the interval [0,1] by two shape parameters that are positive and denoted by α and β. Parameter α is related to the number of heads and β to the number of tails, so $p(\theta) \sim B(\alpha, \beta)$. The probability density function of Beta is:

$$\begin{aligned} p(\theta|\alpha,\beta) &= \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)} \\ &\propto \theta^{\alpha-1}(1-\theta)^{\beta-1}. \end{aligned} \tag{2.7}$$

We can estimate the hyperparameters α and β from $\mu = \frac{\alpha}{\alpha+\beta}$ and $\sigma^2 = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$. So in our example, $\alpha = 13.69$ and $\beta = 5.87$ and the prior $p(\theta) \sim B(13.69, 5.87)$.

Step 2: Define the likelihood/sampling model for the data.

The likelihood for flipping the coin several times is expressed in terms of a Binomial distribution since this distribution is used when dealing with a sequence of n independent experiments, Bernoulli trials, with an expected outcome of either success or failure. Hence, the likelihood is $Bin(H|n, \theta)$, with H denoting the number of heads and n the repetitions of the experiment.

The Binomial likelihood is:

$$\begin{aligned} p(H|n,\theta) &= \binom{n}{H}\theta^{H}(1-\theta)^{n-H} \\ &\propto \theta^{H}(1-\theta)^{n-H}. \end{aligned} \tag{2.8}$$

Step 3: Define the posterior distributions of the unknown parameter.

We apply Bayes' theorem (2.4) to update our beliefs about the parameter θ and derive the posterior distribution.

$$\begin{aligned} p(\theta|H,n) = \frac{p(\theta)p(H|n,\theta)}{p(n,H)} &= \frac{p(\theta)Bin(H|n,\theta)}{\int_0^1 p(H,n|\theta)p(\theta)d\theta} \\ &\propto \theta^{a-1}(1-\theta)^{\beta-1}\theta^{H}(1-\theta)^{n-H} \\ &\propto \theta^{H+a-1}(1-\theta)^{n-H+b-1}. \end{aligned}$$

So, the posterior distribution $p(\theta|H,n)$ is a Beta distribution $B(H+\alpha, n-H+\beta)$.

Suppose that we flip a coin 200 times and we observe 120 heads. The prior is $p(\theta) \sim B(13.69, 5.87)$ and based on the third step, the posterior becomes $p(\theta|H) \sim B(133.69, 85.87)$. A graphical representation of the prior and the posterior distributions of this example is shown in Figure 2.1.

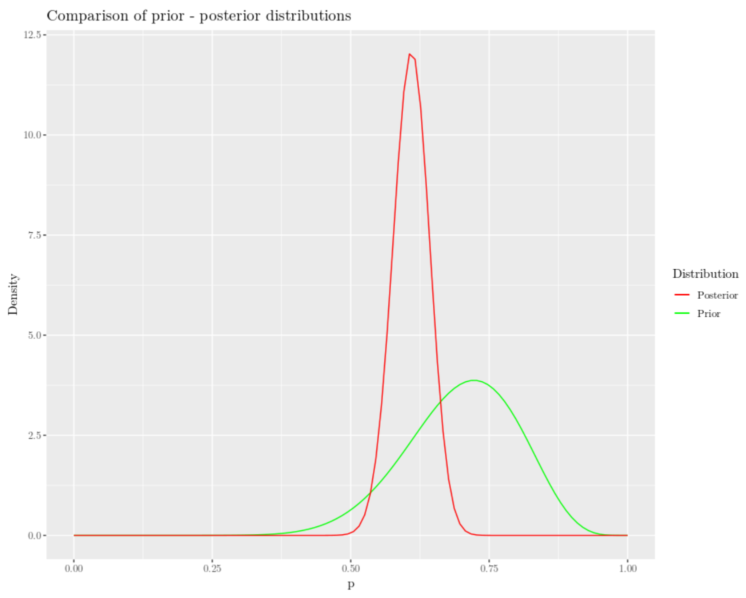


Figure 2.1: Prior and posterior distributions for the biased coin toss example

If a new experiment is run, the new posterior distribution can be formulated by using the previous posterior as a prior distribution and by simply incorporating the new information into the model by adding the new number of "successes/heads" and "failures/tails" to α and β as in the previous example.

2.1.3 Integration

In the typical case of continuous parameters the Bayesian paradigm is based on integration. It is an important part of the estimation of posterior distributions and measurement of uncertainty, as well as of the prediction procedure and model comparison (Angelino *et al.*, 2016).

When the unknown parameter is continuous the calculation of the denominator of Bayes' rule (2.5) involves solving an integral to marginalise the dependence on the unknown parameters out of the joint distribution. In the simple case we only consider one continuous parameter. However, real life problems are usually set in a multidimensional space and involve both continuous and discrete variables. Then, the integral in the denominator becomes high-dimensional. In many cases, exact integration is difficult, computationally intensive in big data problems or even analytically intractable. Calculating summary statistics of high-dimensional distributions can also require calculations of complex integrals.

In the Bayesian framework there are two main ways to overcome this difficulty. The first is the use of conjugate priors that lead to posterior distributions of known forms. The second is using approximation algorithms, such as the Markov Chain Monte Carlo family, which are often adopted since they use an empirical distribution estimated as the target posterior distribution and they avoid calculating complex integrals. These concepts will be further explained in Sections 2.1.4 and 2.2 respectively.

2.1.4 Conjugate prior distributions

One way to overcome the difficulty of calculating multidimensional integrals to get posterior distributions is to use pairs of likelihood and prior distributions whose combinations have convenient properties and lead to predetermined posterior distributions of known forms. These priors that can be combined with specific likelihoods and result in posteriors that are fixed and belong in the same family of distributions as the prior are called *conjugate* priors. Conjugacy is a property mentioned quite often in Bayesian statistics.

Definition. If $\mathcal{F}$ is a class of sampling distributions $p(y|\theta)$, and $\mathcal{P}$ is a class of prior distributions for θ, then the class $\mathcal{P}$ is conjugate for $\mathcal{F}$ if

$$p(\theta|y) \in \mathcal{P} \text{ for all } p(\cdot|\theta) \in \mathcal{F} \text{ and } p(\cdot) \in \mathcal{P} \text{ (Gelman \textit{et al.}, 2013).}$$

In the example of the coin toss presented in Section 2.1.2.2, using a Binomial likelihood with a Beta prior distribution led to a Beta posterior distribution. Beta distribution is a known conjugate prior for the Binomial distribution.

Conjugate priors can lead to simplified calculations and faster Bayesian updates. However, they restrict the prior-likelihood pairs available, hence they are not ideal for all problems and particularly for large and complex models.

2.2 Markov Chain Monte Carlo

Bayes' rule, which is at the epicentre of the Bayesian paradigm, requires evaluation of multidimensional integrals that arise in the denominator, in order to acquire a proper posterior distribution. For complex problems, these integrals are difficult to calculate. Summarising the posterior distribution with moments, such as the expectation $\mathbb{E}[X]$ of a random variable X, also requires solving often intractable integrals. The Markov Chain Monte Carlo (MCMC) class of algorithms aims to overcome this problem by approximating the target posterior distribution through repetitive sampling.

The idea behind MCMC is to draw samples from the target density and compute the average of the set of samples $\{\theta^{(i)}\}_{i=0}^{n}$. Based on the "Strong law of large numbers" the sample average converges to the expectation μ, as the number of samples tends to infinity, $n \to \infty$. This set is a *discrete time stochastic process* where i represents the time points. The *state space* S is such that $\theta^{(i)} \in S$ and can be either continuous or discrete. At each time point, the generated sample $\theta^{(i)}$ depends only on its previous state, so the series of samples forms a *first order Markov Chain.*

Definition. A first order Markov chain is a stochastic process with the property that the future states are independent of the past states given the present state i. For $A \subseteq S$, where $S \subset R$,

$$P(\theta^{(i+1)} \in A|\theta^{(i)} = x_i, ..., \theta^{(0)} = x_0) = P(x^{(i+1)} \in A|\theta^{(i)} = x_i), \forall x \in S.$$

If this property is independent of time and holds $\forall i$ the Markov chain is called *homogeneous.* In the case of discrete θ with finite state space, P is the *transition matrix* containing the probabilities of moving from a point $x^{(i)}$ to the point $x^{(i+1)}$. So,

$$P(x, y) = P(\theta^{(i+1)} = y|\theta^{(i)} = x).$$

In the continuous case, P is a transition kernel with the conditional cumulative distribution function defined by

$$P(x, y) = P(\theta^{(i+1)} \leq y|\theta^{(i)} = x),$$

and the corresponding conditional density is

$$p(x, y) = \frac{\partial}{\partial y} P(x, y), \ \forall x, y \in S.$$

When a Markov chain runs for a sufficiently long time and results in points of which the probability distribution remains unchanged as time progresses, the chain has reached its *stationary or equilibrium distribution.* Not all Markov Chains have stationary distributions, but all Markov chains that are used in MCMC algorithms do.

Definition. A distribution π is called a *stationary distribution* of a homogeneous Markov chain with transition matrix P if

$$\begin{aligned} \pi(y) &= \sum_S \pi(x)P(x,y) \Leftrightarrow \boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P} && \text{for discrete state space,} \\ \pi(y) &= \int_S \pi(x)p(x,y)dx && \text{for continuous state space.} \end{aligned}$$

In MCMC methods the desired stationary distribution is the posterior distribution that we cannot calculate analytically.

Convergence of Markov chains and validation of their fundamental properties are technical topics explained in detail in plenty of literature resources, for example see Gamerman and Lopes (2006). Here, we will refer to two important properties of the Markov chains called *reversibility* and the *detailed balance criterion.*

Definition. A Markov chain $\theta^{(0)}, \theta^{(1)}, ..., \theta^{(n)}$ is *reversible* if the reversed sequence of states, $\theta^{(n)}, \theta^{(n-1)}, ..., \theta^{(0)}$, which is also a Markov chain, retains the same transition kernel at stationarity.

When a chain is reversible and has reached its stationary distribution, in the discrete state space, the transition matrix of the reversed chain is the same as the transition matrix of the forward chain. Equivalently, the density forms of the transition kernel for both chains are the same in the continuous space.

Definition. The *detailed balance criterion* is held when the equations $P(\theta_{k+1}|\theta_k)\pi(\theta_k) = P(\theta_k|\theta_{k+1})\pi(\theta_{k+1})$ are satisfied for a target distribution $\pi(\theta)$.

Hence, the joint density of a transition from a point θ_{k+1} to θ_k, is the same as the joint density of the opposite transition $\theta_k \to \theta_{k+1}$. If the detailed balance condition is held, it can be shown that the target distribution is the stationary and that the chain is reversible. These properties also apply when multiple chains are used during the MCMC simulation.

There are several variations of MCMC algorithms and the choice of the appropriate method is based on the problem and the application area. Some of the most widely used are the Metropolis–Hastings (MH), the Gibbs sampler and the Hamiltonian Monte Carlo. Over the next parts of this section, we will present these three predominant algorithms and examine the methods used in evaluating their convergence. Finally, we will discuss the role of using a burn-in period and thinning.

2.2.1 Random walk Metropolis and Metropolis–Hastings

One of the most commonly used MCMC algorithms, is the Metropolis–Hastings. We will begin by introducing the most basic form of the routine, the “Random walk Metropolis”.

Random walk Metropolis is a way to generate N number of samples from the posterior distribution by using the un-normalised density, the numerator of Bayes’ rule. The aim is that the sampler, starting from a random point drawn from an initial distribution, will explore adequately the whole posterior space. In addition, based on the shape of the distribution defined by the product of the likelihood and the prior, the frequency of the sampled points will be proportional to the values of the posterior distribution. Therefore, values at the peaks of the distribution will be sampled more often than values at the other parts.

Algorithm 1 provides the details of the basic Metropolis algorithm, supposing that θ is the parameter of interest that we want to estimate.

Algorithm 1 The Metropolis algorithm

```
1. Sample $\theta^0$ from a starting distribution $\theta^0 \sim \pi(\theta)$.
2. for iteration $t = 1$ to $N$, do
     a. Sample a new position to step $\theta^*$ from a proposal distribution at time $t$,
   $J(\theta^*|\theta_{t-1})$
     b. Calculate the acceptance ratio $r = \frac{p(\theta^*|y)}{p(\theta_{t-1}|y)}$
     c. Sample $u \sim Uniform(0,1)$
     d. if $u < r$ then
          Accept the proposal: $\theta_t \leftarrow \theta^*$ with probability min(1,r)
        else
          Reject the proposal: $\theta_t \leftarrow \theta_{t-1}$
        end if
   end for
```

The first step creates an initial point, sampled from a starting distribution $\pi(\theta)$. The second step involves numerous iterations, N. At each iteration, a new position is sampled from the *proposal*, also known as *jumping*, distribution J (step 2a). The acceptance rate is calculated using the ratio of the product of the prior and the likelihood evaluated at the new proposed value over the previous (step 2b). If the proposed value has less relative height than the old, r becomes less than 1. Next, we probabilistically decide whether we will accept the new value with probability r. We sample a random value u from a Uniform distribution (step 2c) and by comparing u to r (step 2d) the proposed value is either accepted or rejected. If the new value has a higher or equal relative height than the old, r

becomes greater or equal to 1, and after steps c and d, the proposed value is accepted with probability 1. Following the same steps, the algorithm can extend to a large number of parameters, where each parameter will have its own proposal distribution. Furthermore, the same process applies at each chain when multiple chains are used. However, it is important to initialise each chain at dispersed parts of the parameter space.

The above algorithm works when the proposal distribution at step 2, is symmetric, so that the probability $J(\theta_t|\theta_{t-1}) = J(\theta_{t-1}|\theta_t)$. This property allows the sampler to reverse its previous step in the parameter space. In the Metropolis algorithm the jumping kernel is usually a normal distribution centred around the current point $N(\theta_{t-1}, \sigma^2)$ with the variance σ^2 determining the step size of the jumping rule. This step size is also called the tuning parameter and should be optimised, for the algorithm to explore adequately and efficiently the parameter space. If σ^2 is small a lot of steps are accepted and the process is time consuming. On the other hand, a big step size leads to numerous low probability proposals and inefficient exploration of the parameter space. Therefore, the optimal stepsize should not only propose values that tend to be accepted within a reasonable amount of time, but also, it should let the algorithm move to relatively long distances in the parameter space. It is mentioned in the literature that a step size that results to an acceptance rate approximately 0.44 is considered to be a good jumping step for univariate (Roberts and Rosenthal, 2009) and 0.23 for multivariate proposals (Roberts *et al.*, 1997).

The Metropolis algorithm can be used when the proposal distribution is symmetric. However, when the symmetry is violated, the sampler can lead to biased samples. Occasions when the nature of the parameters would require an asymmetric proposal distribution, such as a log-Normal, could be to achieve skewness towards higher values, or impose some other constraints, e.g. only positive values. Hence there is a need for an algorithm that can handle the asymmetry.

The Metropolis–Hastings algorithm, is a generalisation of the simple Metropolis algorithm that allows for the use of asymmetric proposal densities and overcomes the problems arising with the basic Metropolis when the distributions are not symmetric. It can also be used when a conjugate prior is unavailable. The Metropolis–Hastings has the same logic as the Metropolis, except for adapting the acceptance ratio in order to correct the asymmetry. An extra fraction is used in the acceptance ratio estimated in the second step of the basic Metropolis algorithm (Algorithm 1, step 2b), to correct for the asymmetry in the jumping rule. The acceptance ratio r becomes

$$r = \frac{p(\theta^*|y)}{p(\theta_{t-1}|y)} \times \frac{J(\theta_{t-1}|\theta^*)}{J(\theta^*|\theta_{t-1})}. \tag{2.9}$$

$J(\theta_{t-1}|\theta^*)$ is the probability that we start at θ^* and move to θ_{t-1} and $J(\theta^*|\theta_{t-1})$ is the probability that we start at θ_{t-1} and jump to θ^*. The transition kernel, or transition distribution T, at time t, is then $T(\theta_t|\theta_{t-1}) = J(\theta_t|\theta_{t-1})\alpha(\theta_{t-1}, \theta_t)$. It is, therefore, a mixture of the jumping distribution with a point mass at θ_{t-1}, weighted by the acceptance rate.

It is interesting to see how the detailed balance criterion is fulfilled in the case of Metropolis–Hastings, and how the empirical distribution of the produced samples $\theta_0, \dots, \theta_N$ approaches the target posterior distribution.

At every step denoted by t, with θ_t representing the value of θ at the beginning of the step and θ_t+1 denoting the value at the end of the step, the detailed balance condition for a parameter θ, if $P(\theta)$ is its stationary distribution, is $T(\theta_{t+1}|\theta_t)P(\theta_t) = T(\theta_t|\theta_{t+1})P(\theta_{t+1})$. For the Metropolis–Hastings we want to check whether the posterior distribution $p(\theta|y)$ is the stationary distribution, so we want to check

$$T(\theta_{t+1}|\theta_t)p(\theta_t|y) = T(\theta_t|\theta_{t+1})p(\theta_{t+1}|y). \tag{2.10}$$

Given that

$$T(\theta_{t+1}|\theta_t) = J(\theta_{t+1}|\theta_t)\alpha(\theta_t, \theta_{t+1}),$$

and that acceptance probability $a(\theta_t, \theta^*)$ for a proposed value θ^* is defined as

$$a(\theta_t, \theta^*) = min\left(1, \frac{p(\theta^*|y)J(\theta_t|\theta^*)}{p(\theta_t|y)J(\theta^*|\theta_t)}\right),$$

then at any step there are two possible outcomes; either the proposal θ^* is accepted and $\theta_{t+1} = \theta^*$, hence $\theta_{t+1} \neq \theta_t$ with transition density $T(\theta_{t+1}|\theta_t)$, or the proposal is rejected, and $\theta_{t+1} = \theta_t$, with transition probability $T(\theta_t|\theta_t)$. The density of the first outcome is:

$$\begin{aligned} T(\theta_{t+1}|\theta_t) &= J(\theta_{t+1}|\theta_t)a(\theta_t, \theta_{t+1}) \\ &= J(\theta^*|\theta_t)a(\theta_t, \theta^*). \end{aligned}$$

The probability of the second outcome, where the chain remains at θ is given by

$$T(\theta_t|\theta_t) = 1 - \int_{\theta^*} J(\theta^*|\theta_t)a(\theta_t, \theta^*)d\theta^*,$$

where the right side of the equation is the probability of rejection, thus one minus the probability that the chain moves to the proposed value θ^*. In that case, since $\theta_{t+1} = \theta_t$, the detailed balance criterion is therefore fulfilled (Gamerman and Lopes, 2006).

In the first case, where we accept the proposal and $\theta_{t+1} \neq \theta_t$, the detailed balance condition is again satisfied since

$$\begin{aligned}T(\theta_{t+1}|\theta_t)p(\theta_t|y) &= p(\theta_t|y)J(\theta_{t+1}|\theta_t)min\left(1, \frac{p(\theta_{t+1}|y)J(\theta_t|\theta_{t+1})}{p(\theta_t|y)J(\theta_{t+1}|\theta_t)}\right)\\ &= min\left(p(\theta_t|y)J(\theta_{t+1}|\theta_t), p(\theta_t|y)J(\theta_{t+1}|\theta_t)\right)\\ &= T(\theta_t|\theta_{t+1})p(\theta_{t+1}|y).\end{aligned}$$

So, the detailed balance criterion 2.10 is fulfilled and $P(\theta) = p(\theta|y)$ is a stationary distribution.

The Random walk Metropolis and the Metropolis–Hastings are frequently used as they can be applied to a broad spectrum of problems. However, they involve a number of rejections. A special case of a Metropolis–Hastings update is called Gibbs update and is always accepted. The Gibbs sampler that uses a Gibbs update can be applied in a wide range of problems, and will be presented in the following part.

2.2.2 Gibbs sampler

The Gibbs sampler is another iterative algorithm of the Markov Chain Monte Carlo family that uses sampling to approximate the target posterior distribution. As opposed to Metropolis–Hastings, the proposed values of this algorithm always have an acceptance probability of one. It can be used when the available prior distributions are semi-conjugate. It is based on sampling from the full conditional distribution of every parameter, which is the probability distribution of the parameter conditioned on the values of all the other unknown variables. Analytical calculation is required to extract the full conditional distributions, which can be difficult for large and complex models.

Supposing that:

- $\boldsymbol{\theta}$ is the vector of parameters for which we want to estimate the posterior distribution: $\boldsymbol{\theta} = \{\theta_1, ..., \theta_p\}$,
- y is the vector of data $y = \{y_1, ..., y_n\}$,
- the joint posterior probability of interest is: $p(\theta_1, ..., \theta_p|y_1, ..., y_n)$,

the Gibbs sampler algorithm is defined as follows:

Algorithm 2 Gibbs sampler

```
1. Choose a starting point $\boldsymbol{\theta}^{(0)} = \{{\theta_1}^{(0)}, ..., {\theta_p}^{(0)}\}$

2. for iteration $i = 1$ to $N$, do
       1. sample ${\theta_1}^{(i)}$ from $p\left(\theta_1 | {\theta_2}^{(i-1)}, {\theta_2}^{(i-1)}, ..., {\theta_p}^{(i-1)}, y\right)$
       2. sample ${\theta_2}^{(i)}$ from $p\left(\theta_2 | {\theta_1}^{(i)}, {\theta_3}^{(i-1)}, ..., {\theta_p}^{(i-1)}, y\right)$
       ⋮
       p. sample ${\theta_p}^{(i)}$ from $p\left(\theta_p | {\theta_1}^{(i)}, {\theta_2}^{(i)}, ..., {\theta_{p-1}}^{(i)}, y\right)$

   end for
```

Starting at a random point for all variables, at every iteration we sample a value for each parameter from its full conditional distribution, using the latest up-to-date variable values when needed. The result of each iteration i is a vector with the updated values for all the parameters $\boldsymbol{\theta}^{(i)} = \{{\theta_1}^{(i)}, ..., {\theta_p}^{(i)}\}$. Consequently, the result after N iterations of the Gibbs sampler is a Markov chain of N dependent sequences of vectors of the estimated parameters, where each sequence $\boldsymbol{\theta}^{(i)}$ depends directly only on its previous one $\boldsymbol{\theta}^{(i-1)}$. The sampling distribution of $\boldsymbol{\theta}^{(n)}$ approaches the target distribution and it is proved that for the Gibbs sampler, each component satisfies the detailed balance criterion, even though the full sampler does not.

Usually, in a Gibbs sampler unknown parameters of high-dimensional problems are updated singly and sequentially. However, it is possible to identify relations and potential dependencies between parameters, group them and update them simultaneously in blocks. This tactic can lead to faster mixing and better performance.

There are statistical problems where some full conditionals do not belong to known families of distributions. When we use a Gibbs sampler and some variables have full conditionals that follow a distribution of a non-standard form, it is a common practice to use Metropolis steps for updating these parameters, and standard Gibbs steps for the ones which the full conditionals can be sampled directly. This combination of the two algorithms is known as *Metropolis-within-Gibbs*.

Even though Metropolis–Hastings and the Gibbs sampler are popular choices among MCMC algorithms, they can sometimes explore inefficiently the posterior space. Hamiltonian Monte Carlo is the next algorithm that we examine. It tends to explore more areas of high probability density and therefore is considered to be more efficient than the other approaches.

2.2.3 Hamiltonian Monte Carlo

At the core of Bayesian statistics is computing expectations of functions of interest, usually of joint posterior distributions, using integration. Since these integrals tend to be difficult and complex to solve, due to the nature of the target distributions, we tend to use numerical approximations to evaluate them. Ideally, we would like these approximations to focus on the parts of the function that contribute more to the expectations, rather than at every part of it. MH and the Gibbs sampler show a random walk behaviour in their updates. An algorithm that would suppress the local random walk behaviour by taking into account the geometry of the posterior distribution could be more efficient in some problems than its counterparts.

Hamiltonian Monte Carlo (HMC) is an algorithm that belongs in the Monte Carlo family and it constitutes a special case of MH. HMC is inspired by an area of physics known as Hamiltonian dynamics. Usually, when presenting HMC analogies to physical systems are used in order to make the notion behind the algorithm more comprehensible. Therefore, the mathematical concept of the algorithm is also equivalent to the physical system of a planet and its gravitational field, and the orbit of other planets and objects moving around it. It can be conceptualised using a vector field were the vectors have the appropriate direction, in an analogous way to the orbits of objects around a planet, they are pulled by gravity but also at the same time they do not crash on the planet due to their momentum. The typical set, which is the area where the probability mass lies, is then a trajectory around the mode, in the same way as there is a physical trajectory of the object around the planet (Betancourt, 2018*a*).

One of the main ideas behind HMC is that instead of the typical posterior space considered in other MCMC algorithms, we use the negative logarithmic transformation of it. This space is called *negative log of the un-normalised posterior density*. Hence, the peaks of the initial posterior space become the lowest parts of the transformed posterior space. Due to "gravitational" forces the sampler will spend more time in these low areas that actually have more probability mass concentrated. Also, due to a "momentum" variable it will be able to explore the other parts of the posterior space.

HMC starts every iteration by adding a corresponding momentum m, with $m \sim MVN(0, M)$, to every parameter θ, so that $\theta \to (m, \theta)$, resulting to an extended parameter space. Then, the target posterior distribution is lifted to include the momenta $p(y, \theta) \to p(m, y, \theta) = p(m|\theta, y)p(y|\theta)p(\theta)$. Next, HMC defines the Hamiltonian function

(H) from the joint density over the extended parameter space.

$$\begin{aligned} H(m,\theta) &= -\log \pi(m,\theta,y) \\ &= \underbrace{-\log p(m|\theta,y)}_{\substack{\text{T} \\ \text{“kinetic energy”}}} \underbrace{-\log p(y|\theta)p(\theta)}_{\substack{\text{V} \\ \text{“potential energy”}}}. \end{aligned}$$

The kinetic energy is not specified by the target density, as happens with the potential energy. It is the tuning parameter. Therefore some choices of kinetic energy are more effective than others. The best choice of the kinetic energy factor is an issue that the latest research in the field of HMC, is trying to explore and tackle.

From the Hamiltonian function, the joint system (θ, m) is evolved via Hamilton's equations (Stan, 2019*c*; Neal, 2012):

$$\frac{d\theta}{dt} = +\frac{\partial H}{\partial m} = +\frac{\partial T}{\partial m}$$

$$\frac{dm}{dt} = -\frac{\partial H}{\partial \theta} = -\frac{\partial T}{\partial \theta} - \frac{\partial V}{\partial \theta}.$$

Since the momentum density is independent of the target density, the momentum time derivative is zero $\frac{\partial T}{\partial \theta} = 0$, thus the pair time derivatives become:

$$\frac{d\theta}{dt} = +\frac{\partial T}{\partial m} \quad \text{and} \quad \frac{dm}{dt} = -\frac{\partial V}{\partial \theta}.$$

This system of differential equations can be numerically approximated using a numerical integrator, such as the *Leapfrog integrator* which is specifically adapted for Hamiltonian systems and takes discrete steps with a time interval ϵ. Leapfrog generates the vector field which will show the sampler where the typical set is, and will allow it to move along this typical set and explore the parameter space. In other words it is the path that the sampler has to follow. At each step of the Leapfrog algorithm, the momentum m, initialised from an independent draw $m \sim MVN(0, M)$, gets updated twice, and the position θ is updated once. The result of L repetitions is a pair (m^*, θ^*) that is then used in a Metropolis acceptance step.

The last part is the Metropolis acceptance step and it is very important. By applying the acceptance step, we account for numerical errors produced by the imperfect Leapfrog integrator. The probability of keeping the proposal (m^*, θ^*) generated by transitioning from the previous values (m, θ) is defined as:

$$\min(1, \exp(H(m,\theta) - H(m^*,\theta^*)).$$

After exploring the parameter space using the trajectories we can project down to the original parameter space that defines the next value of the parameter θ.

The idea behind Hamiltonian Monte Carlo, as well as the Leapfrog integrator, is presented with algorithmic steps in the following description:

```
**Algorithm 3** Leapfrog Integrator
for iteration $i = 1$ to $L$, do
    1.    $m \sim MVN(0, M)$
    2.    $m \leftarrow m - \frac{\epsilon}{2}\frac{\partial V}{\partial \theta}$
    3.    $\theta \leftarrow \theta + \epsilon M^{-1} m$
    4.    $m \leftarrow m - \frac{\epsilon}{2}\frac{\partial V}{\partial \theta}$
end for
From all the $L$ steps create the proposal: $(m^*, \theta^*)$
```

```
**Algorithm 4** Hamiltonian Monte Carlo
If $\theta$ represents the set of parameters,
for iteration $i = 1$ to $N$, do
    1. Add a momentum for every $\theta$, so that $\theta \to (m, \theta)$,
       $m \sim MVN(0, M)$
    2. Use the joint density to find Hamilton's equations:
       $$\frac{d\theta}{dt} = +\frac{\partial T}{\partial m} \quad \text{and} \quad \frac{dm}{dt} = -\frac{\partial V}{\partial \theta}$$
    3. Using $L$ steps of the leapfrog integrator, solve the two-state differential equation to get the resulting state $(m^*, \theta^*)$
    4. Accept or reject the proposal $(m^*, \theta^*)$ with probability:
       $\pi(accept) = \min(1, \exp(H(m, \theta) - H(m^*, \theta^*))$
end for
```

It is proven that the algorithm preserves detailed balance (Gelman *et al.*, 2013). A version of this algorithm that requires less amount of tuning, is proposed by Wu *et al.* (2015) and it is called emprical HMC, or eHMC. The number of steps is not defined by the user; it is decided based on "the distribution of the integration time of the leapfrog integrator", and more specifically on the "empirical distribution of the longest integration times".

HMC has the ability to explore the posterior space more efficiently for some models, compared to other MCMC algorithms. Thus, it has attracted considerable scientific interest over the last years.

2.2.4 Convergence and metrics

We saw in the previous sections that MCMC methods are iterative algorithms. Numerous iterations have a computational and time cost. Therefore, there are some challenges arising, such as the number of iterations that the algorithm needs to run for, in order to achieve convergence and secure certainty that the resulting distributions actually are close approximations of the true posterior.

The validity of the MCMC results is based on the assumption that the Markov chain has "converged" to its stationary distribution, so the probability distribution will remain unchanged after that point, after having explored sufficiently the parameter space. Convergence times depend highly on the complexity of the problem as high-dimensional problems tend to need longer run lengths. According to Brooks and Roberts (1998), there are no generic techniques in literature that can be used for *a priori* prediction of run lengths. Therefore, it is important to apply diagnostic tests and convergence diagnostics that provide a better insight about the behaviour of the Markov chain, even though it is not guaranteed that they have successfully diagnosed convergence.

Plotting various diagnostic plots and visually looking for tendencies is a classical approach followed. For example, decaying tendencies in autocorrelation plots (ACF), where the autocorrelation function is plotted against the lag, indicate low serial correlation between successive values in the chain. In addition, posterior samples that have adequate state changes, in particular around the mode(s) of the distribution, and avoid flat regions in the traceplots of the estimated parameters, indicate good mixing. Using a simulation study with synthetic datasets can also be helpful. Even though the distribution of the simulated parameters after conditioning on the data is unknown, the known expected parameter values can be used as an indicator of whether there is a significant problem in the MCMC output. Both methods of using diagnostic plots and synthetic datasets are adopted in Chapter 7.

After comparing various diagnostic methodologies, Brooks and Roberts (1998) conclude that there is not a single best diagnostic method, but the choice of the diagnostic test should depend on the nature of the problem. For example, it is suggested that for unimodal densities the method proposed by Gelman and Rubin is a good approach. The Gelman-Rubin diagnostic evaluates convergence by analysing and comparing the between and within-chain variances for each parameter of the model. Significant differences in these variances are a non-convergence indicator (Gelman *et al.*, 2013). A combination of this method with complimentary techniques can be used for the multimodal cases. Another diagnostic test used when we are interested in posterior quantile values is the Raftery-Lewis. It calculates the number of iterations and the estimated burn-in period for a required level of accuracy (Raftery and Lewis, 1992). Furthermore, some methods involve repetitive hypothesis testing such as the Heidelberger-Welch diagnostic, that calculates a test statistic after checking samples from the length of the run and accepts or rejects the null hypothesis that the Markov chain formed comes from a stationary distribution (Welch and Heidelberger, 1983).

A more recent approach to evaluate the goodness-of-fit is the *kernel Stein discrepancy*. Oates *et al.* (2016) introduced the combination of Stein's identity and reproducing kernel Hilbert space (RKHS) for variance reduction. This combination was subsequently explored by Liu *et al.* (2016), who were also influenced by Gorham and Mackey (2015) and their Stein discrepancy measure, that uses Stein's method to define under which conditions two smooth densities, $p(x)$ and $q(x)$ supported on $\mathbb{R}$, are identical. Combining these ideas Liu *et al.* (2016) present the kernel Stein Discrepancy (KSD, referred to as "kernelised Stein discrepancy" in their paper), a statistic for evaluating the goodness-of-fit of models without using the likelihood. It constitutes a discrepancy measure between distributions, and tests whether a sample $\{x_i\} \sim p(x)$ is drawn from a distribution $q(x)$, so $Ho : p = q$. Therefore, it can be used to compute the discrepancy between the distribution defined by the MCMC output and the target posterior distribution.

Convergence diagnostics have various trade-offs, some might be less computationally expensive and others might be more generally applicable or more easily interpretable. However, they are important in enhancing our notion about the accuracy of the MCMC outcome.

2.2.5 Burn-in period and thinning

The burn-in period, also seen in literature as the warm-up period, is the early phase of the simulations where the MCMC has not yet reached a high probability region. During the first iterations of the algorithm, it is likely that the sampled values are highly biased to the starting value of the sampler, and the chain represents more the initialisation distribution rather than the target posterior. This sequence of samples, where the algorithm starts from an initial value and gets closer to the mass of the distribution, is usually discarded. There is no rule for the number of first n iterations that will have to be ignored from the observed chain; it depends on the MCMC algorithm and the problem. Using long runs of the MCMC and reasonable burn-in periods increases the probability that the chain will have moved towards a higher probability region.

Thinning an MCMC sequence refers to a sub-sampling technique where every kth observation of the resulting chain is kept and all the intermediate estimates are discarded. In the literature there is sometimes a discussion among Bayesian researchers about the necessity and benefits of thinning. In some cases, the practice of thinning the final chain, the result of a long run, is discouraged as it is considered inefficient and not significant in improving either the estimate-mean of the samples nor potential autocorrelation issues. Using longer runs or multiple chains is suggested instead (Link and Eaton, 2012; MacEachern and Berliner, 1994; Jackman, 2009). Geyer (1991) acknowledges that thinning can improve statistical efficiency, an idea mentioned by Owen as well (Owen, 2017), but who also believes that due to the fast decay of the autocorrelations in the MCMC, using a thinning factor may not add significant value. In any case, it is often supported in literature that thinning can be a strategy to handle the large volume of MCMC output and reduce storage cost (Jackman, 2009; Link and Eaton, 2012). In this project we adopt this approach since we want to achieve MCMC scalability for bigger problems.

Alternative approaches to the common practice of using burn-in period and the typical thinning of every kth element of the MCMC output for compression, have also been proposed. Riabiz *et al.* (2020) present a method that uses minimisation of a kernel Stein discrepancy, also mentioned in Section 2.2.4, to select an index set, with size equal to the desired cardinality, out of all MCMC points such that the empirical approximation is close to optimal. Another method for sub-sampling of the MCMC outcome is proposed by Paige *et al.* (2016). Their approach is based on “Algorithm R”, a reservoir sampling algorithm that initially fills a reservoir Y_M, where $M << N$ and N represents the number of MCMC samples. “Algorithm R” initially fills the reservoir with the first M elements of the stream of MCMC samples and then it randomly replaces elements of the reservoir with elements from the remaining output samples $\{x_{M+1}, \dots, x_N\}$. Their proposed algorithm instead of randomly replacing elements of the reservoir, uses the elements that constitute the subset of the MCMC output that minimises the maximum mean discrepancy (MMD), which represents the distance between the mean embeddings of two distributions in the reproducing kernel Hilbert space.

2.3 Bayesian hierarchical models

An important category of models in Bayesian statistics is called *hierarchical models*. They are used when the data are organised in groups. For this structure, an optimal model returns a single posterior distribution that accounts, not only for the variability among the units within a group, but also the heterogeneity between the different groups. Hierarchical models reflect this structure.

As an example of a basic hierarchical model (Hoff, 2009) we can consider a simple case for which both the within and the between group models follow a normal distribution, described as:

- within group: $p(y|\theta_j, \sigma_\theta^2) \sim N(\theta_j, {\sigma_\theta}^2)$
- between group: $p(\theta_j|\mu, \sigma^2) \sim N(\mu, \sigma^2)$

In hierarchical models usually we assign a prior on the group variability measurement, because we want the model to learn from the data the within-group variation. Then, it is common to use precision instead of variance because of the conjugacy properties arising when using a Gamma prior on precision. Therefore, $\theta_j \sim N(\mu_\theta, 1/\tau_\theta)$ and $\tau_\theta \sim \mathit{\Gamma}(\alpha, \beta)$. In the above description of this simple model, the unknown parameters are the group specific means $\boldsymbol{\theta} = \{\theta_1, ..., \theta_p\}$, the within-group variance ${\sigma_\theta}^2$, which is common for all groups, as well as the population mean and variance, μ and σ^2. We assume that the within-group mean is zero.

In Bayesian statistics, models are usually graphically represented using *Directed Acyclic Graphs* (DAG), also refered to as *Bayesian Networks.* DAGs have different interpretations depending on the field, but in Bayesian statistics they are used to capture conditional independence and hierarchy among the variables. The causal relationships between variables are expressed using directed arrows. Hence, if X is causal to Y, an arrow $X \rightarrow Y$ is used (Martin and Fu, 2019). DAGs are particularly useful for hierarchical models as there might be a multilevel dependency across variables, which is easier to understand when illustrated. The DAG for the simple hierarchical model we present is shown at Figure 2.2,

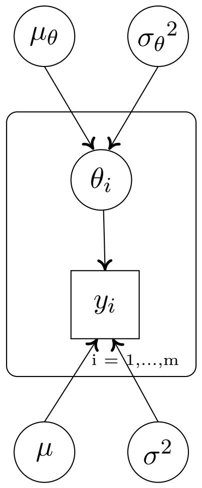


Figure 2.2: Graphical representation of a basic hierarchical model

A more complex example is when we have subgroups within groups. Since in the case of hierarchical models the structure is nested we could imagine Y_{ijk} representing the *i-th* observation within the *j-th* subgroup of the *k-th* group. The correlation between units of the same subgroups or groups is stronger compared to the correlation between units coming from different subgroups or groups (Scott *et al.*, 2013). This happens because hierarchical models account for the possible dependence and commonality between the individual units of groups/subgroups. A model like this is defined as:

$$\begin{aligned}
Y_{ijk} &= \mu + \alpha_k + \beta_{jk} + \epsilon_{ijk}, \text{ where:} \\
\mu &\sim N(\mu_0, v_0) \text{ is the overall mean ,} \\
\alpha_k &\sim N(0, \tau_\alpha) \text{ is the effect of the } k\text{-}th \text{ group ,} \\
\beta_{jk} &\sim N(0, \tau_\beta) \text{ is the effect of the } j\text{-}th \text{ subgroup within the } k\text{-}th \text{ group.}
\end{aligned}$$

Hierarchical models can be evaluated using a Gibbs sampler (2.2.2). In this algorithm, we sample from the full conditionals of each unknown variable in the hierarchical model, in order to approximate the target joint posterior distribution. The process involves defining the prior distributions and the likelihood; multiplying these to result in their joint distribution, from which we can extract the full conditionals of each variable of interest, by isolating only the terms that are related to the variable. Since hierarchical models have multilevel dependencies, DAGs can help us identify the conditional dependencies among the variables and confirm that we extract the correct terms. This process, briefly described here, is analytically presented in Section 3.3. There, we will consider a more complex hierarchical model. We will explicitly define the prior distributions and the likelihood, and we will analytically derive the full conditionals for the parameters of interest, to later implement our own Gibbs sampler.

2.4 Variable selection

In a simple regression problem the aim is to estimate the relationship between the outcome y_i for individual i, $i = 1, ..., N$ and the explanatory variables x_{ij}, with the p covariate weights denoted as β_j and $j = 1, ..., p$. The model describing this relationship is defined as

$$y_i = \alpha + \sum_{j=1}^{p} \theta_j x_{ij} + e_i.$$

Statistical analysis focuses on determining an optimal model that best describes our problem. When a large number of explanatory variables is used to explain the response variable (p is large), we want to identify the subset that best explains the biggest part of the variation in the response. The aim is to identify which θ_js should be set close or equal to zero.

Sometimes, there is *a priori* knowledge reflecting our expectation about the proportion of the covariates that affect the outcome, and that therefore should be included in the model. This information should be taken into consideration in the variable selection. The complexity of the model has some flexibility by defining a prior to the inclusion probability of the covariates. A smaller inclusion probability leads to sparser models. The optimal subset defining the *degree of sparseness*, along with the number of false detections, depends on the problem (O'Hara and Sillanpaa, 2009).

There are many variable selection methods, and we are going to use two basic ways to classify them. The first, is according to O'Hara's and Sillanpaa's categorisation, that is based on how the methods approach various properties (O'Hara and Sillanpaa, 2009). The second is following Piironen's and Vehtari's grouping, based on the choice of prior distributions imposed on the predictor variables (Piironen and Vehtari, 2017). The next two sections focus on presenting these categorisations and methodologies. We particularly focus on two approaches, the Kuo and Mallick and the Horseshoe prior, since they are going to be used later in our hierarchical modelling case study.

The best strategy depends on the the purpose of the study, type of model, and constraints on computational resources. After deciding the structure of the model and the variable selection method, MCMC techniques can be used to estimate the unknown variables.

2.4.1 Variable selection methods

According to O'Hara and Sillanpaa (2009) the main approaches used for variable selection can be classified into four main categories. Each category treats differently important properties such as tuning parameters in prior distributions, and using global or local adaptation according to whether one common variance is used for auxiliary variables or each has its own variance. Most of the prior distributional families used in the following variable selection methods are motivated more by computational than statistical modelling considerations as efficiency is usually of interest, but in most cases the choice of prior parameters can still be scientifically informed. Analytically the four categories are:

1) Indicator model selection

This approach is commonly used, and according to Piironen and Vehtari (2017), often considered as the "golden standard" for Bayesian variable elimination. It is based on the idea of using a "spike and slab" prior distribution on each regression coefficient subject to deletion (George and McCulloch, 1993). The spike represents the probability mass around zero, and the slab refers to the flat shape of the distribution everywhere else. It involves one indicator variable, denoted as I_j ($I_j \in \{0, 1\}$), for each coefficient θ_j. If $I_j = 0$, the coefficient θ_j comes from the "spike" and is close to zero. Conversely, θ_j is nonzero, so it comes from the "slab", if $I_j = 1$.

The "spike and slab" prior can be written as a two-component mixture of Gaussians

(Piironen and Vehtari, 2017)

$$\theta_j | I_j, c, \epsilon \sim I_j N(0, c^2) + (1 - I_j) N(0, \epsilon^2),$$

$$I_j \sim Ber(p),$$

so that, $\epsilon \ll c$, where c is the slab width. The first term is related to the case where the variable is included in the model, and the second to when it is not.

In the indicator model selection approach another auxiliary variable β_j, associated with each θ_j, is also used to represent the effect size, with the final effect of every coefficient being their product $\theta_j = I_j \beta_j$. Then, the slab is defined as $\theta_j | (I_j = 1)$ and is set equal to β_j, and the spike $\theta_j | (I_j = 0)$ equal to 0. There are two main methods implementing this approach, Kuo and Mallick, and Gibbs variable selection (GVS).

1a) Kuo and Mallick

In this method, for every coefficient θ_j there is a variable selection indicator I_j, and the effect of every coefficient is their product $\theta_j = I_j \beta_j$. Independent priors are placed on each I_j and β_j, so $P(I_j, \beta_j) = P(I_j)P(\beta_j)$. Updating the values of θ_j is achieved using MCMC. First, we sample I_j if it is equal to one, we also sample β_j from its full conditional. If $I_j = 0$, we draw a sample for β_j from its prior distribution. The disadvantage of this method is that using a too vague prior might lead to poor mixing because the sampler will not flipping from $I_j = 0$ to $I_j = 1$ often. This approach incorporates all 2^p possible submodels and the marginal posterior distribution of each selection indicator measures the likelihood of each variable/submodel to be included in the model (Kuo and Mallick, 1998). On average we visit each model proportionally to its posterior distribution.

In this project we are implementing and exploring this approach, to entail some form of dimensionality reduction and identify the subset of variables that should be included in the model. In Section 3.3.2, we explain in detail how the Kuo and Mallick method can be embedded in a particular hierarchical model that we examine. We also show how the full conditionals are derived for all the parameters of the model, including the auxiliary variables of this variable selection technique, in order to fit the model with dimensionality reduction, using a Gibbs sampler.

1b) Gibbs variable selection (GVS)

Gibbs variable selection uses again the notion of the variable selection indicator I_j, and the effect of every coefficient being $\theta_j = I_j \beta_j$, as in the Kuo and Mallick method. However, when $I_j = 0$, instead of sampling β_j from a too vague prior distribution, it uses a “pseudo-prior”, that has no effect on the posterior. Unlike the previous approach, the prior distributions of the indicator and the effect are not independent, so $P(I_j, \beta_j) = P(\beta_j | I_j)P(I_j)$. The idea is to use a prior for $\beta_j | (I_j = 0)$ that is concentrated around the posterior density of θ so that $P(\beta_j|I_j = 1)$ will be large when $I_j = 0$ that increases the probability that the chain will move to $I_j = 1$. Another difference from the Kuo and Mallick approach is that the GVS requires tuning to determine which values are good for β_j when $I_j = 0$.

2) Stochastic search variable selection (SSVS)

In this method, as in the case of GVS, we assume that the variable selection indicator affects the prior distribution of β_j, for example $P(I_j, \beta_j) = P(\beta_j|I_j)P(I_j)$. The prior for β_j is a mixture of the type: $P(I_j, \beta_j) = (1-I_j)N(0, \tau^2)+I_jN(0, g\tau^2)$. The basic idea is to have a spike centred around zero with prior variances close to zero for the parameters that should be opted out from the model (first term of the mixture of Gaussians). If the prior parameters, τ^2 and gτ^2, are fixed, tuning is necessary. Otherwise, in some multivariate cases, these parameters can also vary and have a prior distribution.

3) Adaptive shrinkage

The idea of this methodology is to assign directly a prior on θ in order to approximate the "slab" and "spike", instead of using auxiliary indicator variables. Consequently, $\theta_j = \beta_j$ and the prior for β_j is $(\beta_j|\tau^2 \sim N(0, \tau^2))$. A prior is then placed on τ^2 as well. The degree of sparseness can be adjusted by changing the prior distribution of τ^2. Based on the evidence in the data τ^2 will shrink towards zero the values of the covariates β_j that should not be included in the model. Sometimes, τ^2, is assigned a *Jeffrey's prior*, so that P(τ^2)$\sim \frac{1}{\tau^2}$. In the following Section 3.3.3, we are going to look in more detail at a specific continuous shrinkage prior that can be used for handling sparsity, called the *Horseshoe prior.*

4) Model space approach

In contrast with the methods mentioned so far, that place priors on each coefficient θ_j, this approach sees the model as a whole. It splits the problem of variable selection into two main steps. Firstly, it places a prior distribution on the number of explanatory variables selected to be in the model, denoted as N_v with the corresponding vector of coefficients θ_m, $m = 1, ..., N_v$. The second step is to identify which covariates will eventually be selected to be included in the model.

One of the methods that fall into this category is *Reversible jump MCMC.* This method randomly selects a variable j and using Metropolis–Hastings proposes whether it should be added to the numbers of the variables included or not. Controlling the degree of sparseness is achieved through adjusting the prior for N_v. Furthermore, setting a binomial prior on N_v approximates the approach of using independent priors for each I_j. However, the disadvantage of the algorithm is that it can show increased complexity due to the change in dimensions, with new variables being constantly added or removed. Setting a maximum number of covariates that can be included in the model can restrict the size from escalating and causing efficiency and complexity issues.

At this point it would be useful to mention that according to Johnson and Rossell (2012), most of the Bayesian variable selection methods for linear models use local prior densities for the regression coefficients, that place a significant mass in the vicinity of zero to the parameters that should be included in the model. Their suggested non-local prior approach penalises models that include components that are equal to zero, in contrast to classical Bayesian methods that assign positive density values to these models. They support that for large sample problems, using the two non-local prior densities proposed can not only lead to identification of the most probable model, but also to estimates of the posterior probability that each model is correct, which constitutes a weakness of the common Bayesian variable selection procedures. So, in the four categories of techniques mentioned above, non-local prior distributions could be used as an alternative to local priors centred on zero.

2.4.2 Prior choices and the Horseshoe prior

Another way to differentiate between the methodologies used for Bayesian variable selection is the choice of priors. Different methods take advantage of different specific characteristics of priors. The categories of priors, can be separated into two main groups: two component discrete mixture priors, and a variety of continuous shrinkage priors (Carvalho *et al.*, 2009; Piironen and Vehtari, 2017).

The first approach of discrete mixtures, also referred to as spike-and-slab was presented in the previous section (2.4.1, part 1). The second, was briefly introduced earlier (2.4.1, part 3), and will be explained more analytically in this section, using an interesting case of the continuous shrinkage priors, the Horseshoe prior.

The first application of the Horseshoe prior was for sparse signal and noise detection, but it can be applied to various domains (Carvalho *et al.*, 2008). For implementing the Horseshoe, two new variables are introduced, τ (does not relate to precision) and λ. For the simple example of linear regression with N coefficients, the Horseshoe prior can be defined as:

$$\begin{aligned} \beta_j|\lambda_j,\tau &\sim N(0,\tau^2\lambda_j^2), \quad j=1,...,N,\\ \lambda_j &\sim C^+(0,1),\\ \tau &\sim C^+(0,1). \end{aligned} \tag{2.11}$$

Half-Cauchy distributions ($C^+(0,1)$) are used for defining the Horseshoe prior. The probability density function of a Cauchy distribution, with s representing the scale and m the location, is:

$$f(x) = \frac{1}{\pi s} \frac{s^2}{(x-m)^2 + s^2}.$$

Half-Cauchy is defined as the right part of the symmetric halves of Cauchy, corresponding to the positive values of the random variable, with the scale being equal to one and the location to zero. So, its probability density function (pdf) is:

$$f(x) = \frac{2}{\pi(x^2+1)}, \tag{2.12}$$

where $x > 0$.

The Horseshoe prior, is also called a global-local shrinkage prior. "Global" derives from the fact that it uses the global hyperparameter τ that tends strongly to pull all the parameters to zero. On the other hand, the "local" character is defined by the local parameters λ_j, that due to the heavy tails of the half-Cauchy prior distribution, allow for some weights θ_j to escape and not to be shrunk to zero. All λ_js have independent half-Cauchy priors. The flexibility in the degree of sparseness is provided by τ; the smaller the τ the more β_js are shrunk towards zero. Hence, even though the global shrinkage parameter tries to estimate the sparsity overall, the heavy tails of the prior for the local shrinkage parameter enable estimates β_j to be formed separately from τ. Even though a half-Cauchy $C^+(0,1)$ is a standard prior choice for the local parameters, there is discussion about the most appropriate prior on τ, with various alternatives considered (Piironen and Vehtari, 2017). However, it is common practice to assign a half-Cauchy prior to the global hyperparameter as well (Gelman *et al.*, 2013; Carvalho *et al.*, 2009).

For each coefficient β_j, there is an equivalent *shrinkage factor* κ_j (2.13), such that the posterior mean estimate of the coefficient β_j, denoted as $\bar{\beta_j}$ or $E(\beta_j|y_i, \lambda_i{}^2)$, is equal to $\bar{\beta_j} = (1-\kappa_j)\hat{\beta_j}$, where $\hat{\beta_j}$ is the maximum likelihood estimate. The shrinkage factor describes the weight that the posterior mean for β_i places on zero after observing the data (Carvalho *et al.*, 2009) and actually shows how much β_j is shrunk towards zero. It is defined as:

$$\kappa_j = \frac{1}{1 + n\sigma^{-2}\tau^2 s_j^2 \lambda_j^2}, \tag{2.13}$$

where n is the number of observations, σ^2 is the model error and s_j^2 is the variance of the predictors. The closer to one κ_j is, the more shrinkage it indicates and vice versa. From 2.13, we can see that $\bar{\beta_j} \to 0$ as $\tau \to 0$ and $\bar{\beta_j} \to \hat{\beta_j}$ as $\tau \to \infty$ (Piironen and Vehtari, 2017).

Different prior choices on λ_j change the prior distribution of κ_j. When a half-Cauchy is assigned to the local parameters as their prior and under particular assumptions, the shrinkage factor's distribution becomes a Beta $\left(\frac{1}{2}, \frac{1}{2}\right)$, taking the shape shown in Figure 2.3, where the name "Horseshoe" derives from (Carvalho *et al.*, 2009).

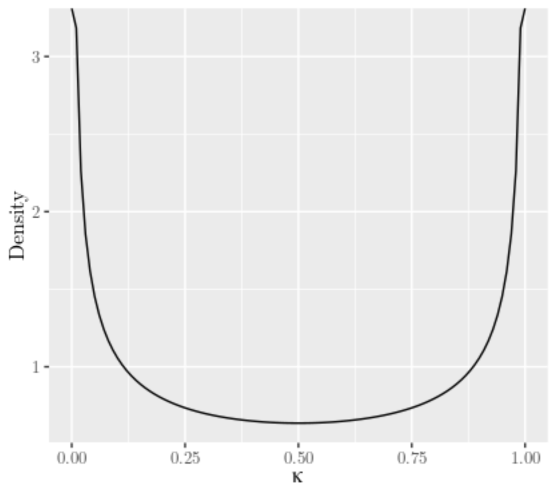


Figure 2.3: Shape of the Horseshoe prior

According to Piironen and Vehtari (2017) the standard Horseshoe approach does not guarantee that the prior will always shrink the coefficients by at least a small amount. Therefore, they introduce the *regularised horseshoe prior* that is defined as:

$$\begin{gathered}\beta_j|\lambda_j, \tau, c \sim N(0, \tau^2\tilde{\lambda}_j^2), \quad \tilde{\lambda}_j^2 = \frac{c^2\lambda_j^2}{c^2 + \tau^2\lambda_j^2}, \\ \lambda_j \sim C^+(0,1), \quad j = 1, ..., N.\end{gathered} \tag{2.14}$$

In the case where $\tau^2\lambda_j^2 << c^2$, the coefficient β_j is close to zero, $\tilde{\lambda}_j^2 \to \lambda_j^2$ and the prior approaches the original horseshoe. On the contrary, when $\tau^2\lambda_j^2 >> c^2$, the corresponding coefficient is far from zero. Then, $\tilde{\lambda}_j^2 \to \frac{c^2}{\tau^2}$ and the prior approaches $N(0, c^2)$, so it will also shrink coefficients with larger values. The variance c^2 can have a fixed value, but is usually assigned an inverse-Gamma prior distribution.

The use of continuous shrinkage priors, including the Horseshoe prior, is very important in developing variable selection in the Bayesian framework. These priors allow MCMC algorithms that use integration and need a continuous parameter space, such as the Hamiltonian Monte Carlo, to be used for implementing variable selection. In Section 7.4 we will present analytically how the Horseshoe prior can be incorporated in a two-way Anova variable selection model using a Gibbs sampler.

2.5 Model comparison

In many cases there are various candidate models, out of the *model/hypothesis space* (the set of all possible models), that ascribe the competing hypotheses about how the data were generated. It is then necessary to decide which model is best supported by the available evidence, and therefore, more likely to have generated the observed data, by introducing model comparison and selection.

In Bayesian statistics, model comparison is implemented using either a *Bayes factor*, or one of the available *predictive information criteria*, such as the *Deviance information criterion* (DIC). Before continuing with the presentation of these two approaches, we are going to introduce some of their key concepts, the *marginal likelihoods*, used in the former approach, and the *deviance* and *selection bias* which are important aspects of the latter.

2.5.1 Marginal likelihood

At the core of Bayesian inference is Bayes' rule (2.4), repeated here for convenience:

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)}. \tag{2.15}$$

Typically, in the traditional use of Bayes' rule for inferring the posterior distribution of one or more unknown variables, there is one single model (M_1) that we examine. Therefore, technically in 2.15 every term can also be considered as conditioning on the model M_1. So, Bayes' rule could also be written as:

$$p(\theta|y, M_1) = \frac{p(y|\theta, M_1)p(\theta|M_1)}{p(y|M_1)}. \tag{2.16}$$

The denominator of 2.16 is known as the marginal likelihood of the model or model evidence. It constitutes a likelihood function from which the unknown parameters of the model have been marginalised, with the remaining part representing the probability of the data given the model type, with no assumptions about the underlying specific model parameters. For discrete parameters this marginalisation is achieved through summation. For a continuous parameter vector, marginalisation is attained through integration with respect to the unknown parameters, so that the marginal density does not depend on θ. It conceptualises the likelihood of the observed data under the specific generating process that we consider in the model. In the univariate case, the marginal likelihood can be written as:

$$p(y|M_1) = \int p(y|\theta, M_1)p(\theta|M_1)d\theta. \tag{2.17}$$

The integral in 2.17 becomes a high-dimensional one, when θ is a multiparameter vector, that renders its calculation difficult.

2.5.2 Bayes factor

In the case where two models (M_1 and M_2) are under comparison, the Bayes factor has the underpinning rationale of comparing their posterior distributions, using a ratio of the measures of goodness of fit of the two statistical models. Based on Bayes' rule, the posterior distribution of model M1, given the data is

$$p(M_1|y) = \frac{p(y|M_1)p(M_1)}{p(y)}. \tag{2.18}$$

The numerator consists of the marginal likelihood (2.17) multiplied by the prior distribution of the model ($p(M_1)$). The denominator is the marginal likelihood of the data across all models, so $P(y) = P(y|M_1)P(M_1) + P(y|M_2)P(M_2)$.

After having defined the posterior densities of the models (2.18), the ratio of these densities, known as posterior odds for the model M_1 against M_2 is defined as:

$$\underbrace{\frac{p(M_1|y)}{p(M_2|y)}}_{\substack{\text{Posterior}\\\text{odds}}} = \underbrace{\frac{p(y|M_1)}{p(y|M_2)}}_{\substack{\text{Bayes}\\\text{factor}}} \underbrace{\frac{p(M_1)}{p(M_2)}}_{\substack{\text{Prior}\\\text{odds}}}. \tag{2.19}$$

The Bayes factor in 2.19 is the ratio of the marginal likelihoods for the two models. When the same prior distribution is assigned to both models, their comparison through the posterior odds, is only based on the Bayes factor. A value of that ratio greater than one indicates preference over the model M_1.

Even though the Bayes factor is a known method for model comparison, there are some issues arising with its use for that purpose. Firstly, in complex examples with many unknown parameters, the high-dimensional integrals, necessary for calculating the marginal likelihood, can be intractable. Secondly, marginal likelihoods can be sensitive to prior choices for the unknown parameters, irrespective of the insignificant changes that they may cause to the posterior probabilities of the parameters themselves. Finally, assigning prior probabilities to models can also be difficult (Gelman *et al.*, 2013).

The above addressed issues lead to suggestion of alternative methodologies for model evaluation and comparison, that are based on measurements of expected predictive accuracy which are introduced in the following section.

2.5.3 Information criteria and cross-validation

In determining the parameters of a model, a specific dataset is used. There is then the expectation that the accuracy of the fitting model will be higher for the data coming from this dataset, compared to that of fitting the same model on future out-of-sample observations.

Ideally the predictive accuracy should be its out-of-sample predictive performance, however, out-of-sample data are not always available. Therefore, several measures of estimating the expected predictive accuracy have been developed, that use the same sample that is used to fit the model to. These measures are known as *information criteria* and share two common characteristics; they all include a term corresponding to a measure of fit and a term aiming to account for the selection bias. For a goodness-of-fit measure some criteria are based on the log-likelihood or log predictive density. More precisely, they involve the deviance, which is the log predictive density of the data at a point estimate, multiplied by -2, hence $-2\log(p(y|\hat{\theta})$ [1]. The second common characteristic is the inclusion of a term that will act as a "penalty term" and is necessary to account for the fact that a larger model has a natural ability to fit the data better, which is often referred to as selection bias. This term is a correction for the effective number of parameters (complexity) and adjusts for overfitting. As an alternative to the techniques that use the within-sample predictive accuracy, there is *cross validation* that captures the out-of-sample prediction error by using a training set to fit the data and a test set to evaluate the model's predictive accuracy (Gelman *et al.*, 2013).

The four criteria that share the consensus of within-sample predictive accuracy include a measure to denote the expectation under the true data distribution $f(y)$. This measure is called *expected log predictive predictive density (elpd)*, and, when there is a new data point y^{new}, quantifies how close the estimated posterior predictive distribution $p(y^{new}|\mathbf{y})$, is to the true distribution $f(y)$. More analytically the four criteria are:

1. Akaike information criterion (AIC)

   For this criterion, the *expected log predictive density (elpd)* for a new dataset is estimated using the maximum likelihood estimate, and is given by:

$$\widehat{elpd}_{AIC} = \log p(y|\hat{\theta}_{mle}) - k,$$

   where k is the number of parameters in the model. This term acts as a penalty to the increase in the predictive accuracy due to an increase in the number of parameters. AIC is actually defined as the product of $\widehat{elpd}_{AIC}$ by -2, hence $AIC = -2\log p(y|\hat{\theta}_{mle}) + 2k$ (Gelman *et al.*, 2013).

[1] Deviance has a different definition in the context of generalised linear models, denoting the deviation of a model under exploration, from the saturated model for which the data are fitted perfectly.

This criterion has the disadvantage that it can be mainly used for simpler models, as hierarchical structures tend to affect and more specifically reduce the amount of overfitting. Furthermore, in hierarchical models, it is often unclear what the number of parameters k should be.

2. Deviance information criterion (DIC)

   For this criterion, the expected log pointwise predictive density is now estimated using the posterior mean that maximises the posterior density, $\hat{\theta}_{Bayes} = E(\theta|y)$, as a measure of fit. The bias correction is now data based, two times the variance across all of the posterior draws of the log of the likelihood. This penalty is based on the idea that if there is a lot of uncertainty in the parameter values, then there will be a lot of uncertainty in the fit (measured by the variance), meaning that the model could be too complex, so maybe there is overfitting (Lambert, 2018).

   So, DIC is computed as (Gelman *et al.*, 2013):

$$DIC = -2 \log p(y|\hat{\theta}_{Bayes}) + 4\text{var}_{\text{post}}(\log p(\boldsymbol{y}|\theta)).$$

3. Watanabe - Akaike or widely available information criterion (WAIC)

   The previous two criteria use measures of fit that are point estimates to the parameter values. A measure of fit that would take into account the uncertainty in the value of the unknown parameters estimated, would reflect more the Bayesian logic. WAIC incorporates this uncertainty in the estimation of predictive accuracy.

   The measure of fit in this case is the computed log pointwise predictive density (lppd), that is the average fit of the model across all of the draws of θ representing the posterior uncertainty (Lambert, 2018). The bias correction is similar to the correction term used in DIC, but now it computes the variance separately for each data point of the posterior draws, and then calculates their sums. So WAIC is formulated as (Gelman *et al.*, 2013):

$$WAIC = -2 \sum_{i=1}^{n} \log \left(\frac{1}{S} \sum_{s=1}^{S} p(y_i|\theta^s) \right) + 2 \sum_{i=1}^{n} V_{s=1}^{S} (\log p(y_i|\theta^s)),$$

   where the first term is the computed *log pointwise predictive density (lppd)*.

4. ‘Bayesian’ information criterion (BIC)

   According to Gelman, the name ‘Bayesian’ in this criterion is misleading, and the measure might give poor results for complicated models. Furthermore, unlike the other three criteria, BIC is not intended to predict the out-of-sample performance. However, since it is mentioned in the literature, we also make a reference to it here.

BIC has a predictive measure, and a penalty term, that increases with the sample size, n in order to favour simpler models with fewer observations and parameters. More precisely BIC is defined as: BIC $= -2\log p(y|\hat{\theta}) + k \log n$.

The above techniques try to predict the out-of-sample performance by using the same dataset, used to fit the model. In contrast, a technique called *Leave One Out Cross Validation (LOO-CV)*, splits the data into a training set, and a test/cross validation test. The former set is used to fit the model, whereas the latter, is used to validate the models' predictive performance.

Estimating LOO-CV is a repetitive process, where one data point is used as a validation set, and the remaining data points constitute the training set. This process is repeated n times, as the number of sample size, but can be computationally expensive.

At each iteration of LOO-CV the log posterior predictive density, the out-of-sample predictive fit, is estimated across all samples from the posterior distribution, as (Gelman *et al.*, 2013):

$$lppd_{loo-cv} = \sum_{i=1}^{n} \log p_{post(-i)}(y_i) = \sum_{i=1}^{n} \log \left(\frac{1}{S} \sum_{s=1}^{S} p(y_i|\theta^{is}) \right). \tag{2.20}$$

The bias correction here accounts for the fact that each prediction is conditioned on $n-1$ points and we want to estimate the improvement in predictions if we were conditioning on n data points:

$$b = lppd - \overline{lppd}_{-i}, \tag{2.21}$$

where $\overline{lppd}_{-i}$ is calculated as:

$$\overline{lppd}_{-i} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \log \left(\frac{1}{S} \sum_{s=1}^{S} p(y_i|\theta^{is}) \right). \tag{2.22}$$

So, the bias corrected Bayesian LOO-CV is then $lppd_{cloo-cv} = lppd_{loo-cv} + b$. An estimate of the effective number of parameters is used in order to make comparisons to other methods, calculated as:

$$p_{loo-cv} = \overline{lppd}_{-i} - lppd. \tag{2.23}$$

Vehtari *et al.* (2016) suggest a more tractable approximation to LOO-CV, using *Pareto smoothed importance sampling (PSIS)*. Their approach, PSIS-LOO, provides an improved and more reliable estimate. They fit a Pareto distribution to the upper, potentially long, tail of the distribution of the importance weights, in order to smooth the larger values. Their method is available both in R and Stan packages, to provide an efficient approach to LOO-CV for model comparison.

Each of the techniques mentioned above, from both the within sample and out-of-sample framework, has advantages and disadvantages, and depending on the problem they can return similar results. According to Gelman *et al.* (2013), AIC does not work properly when we assign strong prior distributions, and DIC returns non-sensible results when "the posterior distribution is not well summarised by its mean". According to Lambert (2018), both these criteria are not fully Bayesian as they rely on point estimates. On the other hand, WAIC which is fully Bayesian, relies on partitioning data, that can be problematic for some sorts of models, e.g. related to spatial data. Finally, cross-validation can be computationally demanding. They suggest that for these reasons, Bayesian statisticians sometimes avoid using these predictive comparisons, unless they want to compare very different models, in which case, WAIC and cross validation are recommended. However, DIC is often encountered in the literature as it constitutes a simpler criterion to compute using MCMC samples, hence it remains a popular choice.

2.6 Summary

In this chapter we introduced some of the key points of Bayesian computation. We saw that in the Bayesian framework, our uncertainty about the unknown parameters of a model is expressed through probability distributions. Estimation of these parameters involves complex integrals that can lead to difficult and intractable calculations. The Markov Chain Monte Carlo family of algorithms aims to avoid analytical integral solutions and through repetitive sampling, numerically approximate the quantities of interest by taking advantage of fundamental properties of Markov Chains. MCMC methods have a wide range of applications, including hierarchical modelling, particularly useful when the data have a multilevel structure.

Models often have a large number of covariates. In that case, it is frequent practice to apply dimensionality reduction by identifying the parameters that affect the response the most. Variable selection methods, such as using variable selection indicators and shrinkage prior distributions, are a valuable tool to specify the best subset of predictors to include in the model. The Gibbs sampler, one of the most widespread MCMC algorithms, can be used for variable selection, however, its implementation requires analytical derivation of the full conditional distributions of the unknown parameters.

In the following chapter we will explore a specific form of hierarchical models called two-way Anova. We will also present how the full conditional distributions of the model parameters are derived, in order to use Gibbs sampling to evaluate the model.

Chapter 3

Bayesian analysis of Anova models

Anova models are a form of hierarchical model used when the independent variables are categorical. The term *two-way* Anova is used to describe an Anova model that includes two independent variables. In the following parts, after introducing this type of model design, we will demonstrate the derivation of the full conditionals of the parameters of interest, for an expansion of a basic two-way Anova that involves interactions, random effects and variations in the "symmetry" of main and interaction effects, in order to use a Gibbs sampler to build it. Firstly, we will explore the case where all the predictor variables are included in the model, which is called *saturated.* Then, we will expand and adapt these full conditional distributions to incorporate variable selection, where we will be mainly focused.

3.1 Introduction to Anova

Analysis of variance (Anova) is a statistical technique to test the variation among the population means of groups encoded in the form of discrete categorical predictors. The term "two-way" refers to the fact that there are two independent variables in the model. The response is a continuous variable. In two-way Anova, we are often interested not only in the main effects of each categorical variable, but also in their interactions. The full saturated model includes all the main and interaction effects. In this project we are focusing on a more complex version of the basic two-way Anova model by incorporating interactions and random effects, with further application of variable selection on the interactions. A simple description of a basic two-way Anova model is

$$
\begin{aligned}
x_{ijk} &= \mu + \alpha_i + \beta_j + e_{ijk} && \text{without interactions,} \\
x_{ijk} &= \mu + \alpha_i + \beta_j + \gamma_{ij} + e_{ijk} && \text{with interactions,}
\end{aligned}
$$

where, α denotes the first categorical variable with n_α levels and $i = 1, ..., n_\alpha$, and β represents the second categorical variable with n_β levels and $j = 1, ..., n_\beta$. The interactions between the i^{th} level of the first and the j^{th} of the second independent variable is denoted as γ_{jk}.

In Bayesian statistics, linear Anova models have the statistical power of linear hierarchical models (Section 2.3). This is due to the fact that the categorical variables are seen as batches/groups of coefficients, considered to be "sources of variation". With every Anova model there is an associated Anova table, where each batch of coefficients holds one row of the table. A batch can consist of a single experimental block, or a level/factor of the categorical variables, or even interactions of two or more factors. The hierarchical structure of Anova allows the estimation of not only the individual coefficients but also their variance within each group (Gelman *et al.*, 2013).

An important aspect to consider when fitting an Anova model is the decision upon the use of fixed or random effects. In both cases in the Bayesian framework, we assign a prior distribution on the effects. For fixed effects, the parameters of the prior distribution are "fixed" and the samples are independent. Identifiability problems might arise in this case. Hence, there is not a unique set of parameters to optimise the fit of the model to the given data sample. This issue can be solved using *treatment contrasts* and *sum-to-zero constraints*. The first technique works by setting the first level of a categorical variable as base. Its mean is the intercept, so the means of the other levels reflect the change from the first. Adding a sum to zero constraint means that the effects are constrained to sum to zero. The intercept now represents the overall population mean and each coefficient represents the mean difference between what it expresses and the intercept. On the other hand, random effects are not independent even though they are conditionally independent *a priori*, given the parameters of the distribution of the random effects. They are drawn from a prior distribution with parameters which are not fixed. Given that this distribution is often centred to zero, it is then the variance that is assigned a prior, adding an extra level of hierarchy to the model (Wilkinson, 2014; Hoff, 2009). In literature there are often conflicted suggestions about when to use fixed or random effects models. There is an assumption about the random effects, that in order for this approach to return unbiased estimates, conditional independence of the group-specific effects must apply. As long as this assumption is valid, random effects are considered to be more efficient and to have a stronger predictive power than using fixed effects (Scott *et al.*, 2013).

3.2 Variable selection for Anova models

Gibbs sampling is often selected among MCMC algorithms for fitting Bayesian Anova models. The saturated model of a hierarchical two-way Anova problem involves all the main and all the interaction effects. In a statistical problem where the explanatory categorical variables have a large number of levels, the saturated case can lead to a very big model with many independent variables and interpretational difficulties. As we mentioned in Section 2.4, there are Bayesian dimensionality reduction techniques that identify the predictors considered as the most significant for the model. In this project, we assume that all main effects are present and we only apply variable selection to their interactions, since probably not all of the interaction covariates are useful for predicting the response variable. We choose this implementation because otherwise we could result to a model containing interactions of variables for which there is no main effect, which would be unusual, at least for hierarchical modelling (Gelman *et al.*, 2013).

In order to determine which variable selection approach is more effective and performant in the case of hierarchical two-way Anova models, we will explore both spike-and-slab and the shrinkage priors, which are the two main groups of variable selection methods. The first category is a classical approach in Bayesian statistics and is often combined with Gibbs sampling. The priors of the second category aim to enable variable selection with those MCMC algorithms that do not support discrete variables and can sometimes lead to better mixing and faster convergence (Piironen and Vehtari, 2017). We will specifically emphasise two variable selection techniques, the Kuo and Mallick approach (2.4.1, 1a) from the former group and the Horseshoe prior (2.4.2) from the latter, to see how they are applied to Anova problems.

There are existing probabilistic programming frameworks, such as JAGS for Gibbs sampling and STAN for Hamiltonian Monte Carlo, that will be presented in Chapter 6 and used in this thesis as described in Chapter 7, that enable implementation of both variable selection approaches without requiring analytical algebraic calculations from the user. However, when implementing an MCMC algorithm from the ground up, as is the aim of this thesis, algebraic calculations are necessary. We want to implement a Gibbs sampler in the Scala programming language and at the time of working on this research project there was no such active library. When we presented Gibbs sampler (Section 2.2.2), we saw that it draws samples from the full conditional distributions of the unknown parameters to approximate the target posterior distribution. The same process applies to the various models that we explore in this thesis. Even though it is possible to find analytical calculation of full conditional distributions for the parameters of typical Anova models in the literature and on-line sources, our models are non-standard Anova models. The special cases of symmetry in the effects as explained in Section 3.3 and the additional inclusion of particular variables to enable variable selection using the two techniques that we examine, render our models non-trivial cases. Hence, analytical calculation of the full conditional distributions of the parameters of interest is required.

One disadvantage of not using an already existing probabilistic programming framework is the fact that pen-and-paper calculations are prone to errors. In order to be confident that there are no algebraic errors we firstly verify that the calculated full conditional distributions are of the expected form since we use conjugate priors. Furthermore, we conduct a simulation study (Chapter 7) with synthetic examples and compare the results of our code to the true simulated values of the variables. Additionally, we compare the univariate marginal posterior distributions produced from our code in Scala and long-established JAGS. Another disadvantage of this approach is the fact that alterations in model assumptions, such as changes in prior distributions, tend to be inflexible and impractical to directly implement as they require algebraic recalculations.

In the following section we will demonstrate the derivation of the full conditionals of the unknown parameters that we want to estimate for a hierarchical two-way Anova model. In this model, we account for the interactions, as well as the main effects, all of which are treated as random effects. We will develop both the saturated case and the variable selection on the interaction terms.

3.3 Full conditional distributions for two-way Anova

In the following analysis we will consider some variations of symmetry in the effects. More precisely, we examine four cases: both main and interaction effects being asymmetric, both main and interaction effects being symmetric, only the main effects are symmetric and the interactions are asymmetric, and only the interaction effects are symmetric and the main effects are asymmetric. The reason for this approach will become clear when we explore the real dataset of the yeast genome case study in Section 9.2. Treating the effects in a symmetric/asymmetric way is of scientific importance due to the nature of the genetic experiment.

Suppose that the two independent categorical variables of the model are denoted as α and β, each having $j = 1, ..., n_\alpha$ and $k = 1, ..., n_\beta$ levels. The interaction between α_j and β_k is denoted as γ_{jk}. By using the term "symmetric main effects", we refer to the fact that we assume that the main effects of the two independent categorical variables come from the exact same distribution and that $n_\alpha = n_\beta$ and $\alpha_i = \beta_i$ for all i. By "symmetric interactions", we mean that the effect of the interaction γ_{jk} is treated by the model as the same as γ_{kj}.

3.3.1 Saturated model

In the saturated case, we want to quantify the relationship of each level of the explanatory variables with the response, and we assume that all interaction effects are present in the model. Therefore, we estimate all the main effects for each level of the two categorical variables, as well as all the interactions between them.

Here, we only examine the case where both the main and the interaction effects are considered to be asymmetric. This means that the two groups of main effects are different, and the effect of the interaction γ_{jk} is different from γ_{kj}. All effects are treated as random, hence, we consider that the mean of the prior distribution of the effects is zero ($\mu_\alpha = \mu_\beta = \mu_\gamma = 0$) and we assign a gamma prior on their precision. Based on these characteristics, the model is formed as follows:

Model:

$X_{ijk}|\mu, \alpha_j, \beta_k, \gamma_{jk}, \tau$ ~ N($\mu + \alpha_j + \beta_k + \gamma_{jk}$, τ^{-1}), i=1,...,N_{jk},

$N = \sum\limits_{j=1}^{n_\alpha} \sum\limits_{k=1}^{n_\beta} N_{jk}$

μ ~ N(μ_0,τ_0^{-1})

τ ~ $\Gamma(a, b)$

$\alpha_j|\tau_\alpha$ ~ N($\mu_\alpha, \tau_\alpha^{-1}$), j=1,...,$n_\alpha$

$\beta_k|\tau_\beta$ ~ N($\mu_\beta, \tau_\beta^{-1}$), k=1,...,$n_\beta$

$\gamma_{jk}|\tau_\gamma$ ~ N(μ_γ,τ_γ^{-1})

τ_α ~ $\Gamma(a_{\tau_\alpha}$,$b_{\tau_\alpha})$

τ_β ~ $\Gamma(a_{\tau_\beta}$,$b_{\tau_\beta})$

τ_γ ~ $\Gamma(a_{\tau_\gamma}$,$b_{\tau_\gamma})$

Where α_j is the effect of the j^{th} α, β_k is the effect of the k^{th} β and γ_{jk} is their interaction.

Notation:

n_α is the number of levels of the independent categorical variable α

n_β is the number of levels of variable β

$n_\gamma = n_\alpha \cdot n_\beta$ is the number of levels of interactions

$N_j = \sum\limits_{k=1}^{n_\beta} N_{jk}$ is the number of observations that have α_j

$N_k = \sum\limits_{j=1}^{n_\alpha} N_{jk}$ is the number of observations that have β_k

N_{jk} is the number of observations for which the α is j and the β is k

The corresponding Directed Acyclic Graph (DAG) representing the random variables of the model along with their conditional dependencies is presented in Figure 3.1 in form of a plate diagram. Plate notation is used following the convention for WinBUGS (Spiegelhalter *et al.*, 2003). Repeated parts of the graph are represented using a 'plate' (rectangle) to group variables together. The equivalent indices and their range are denoted at the bottom of each plate.

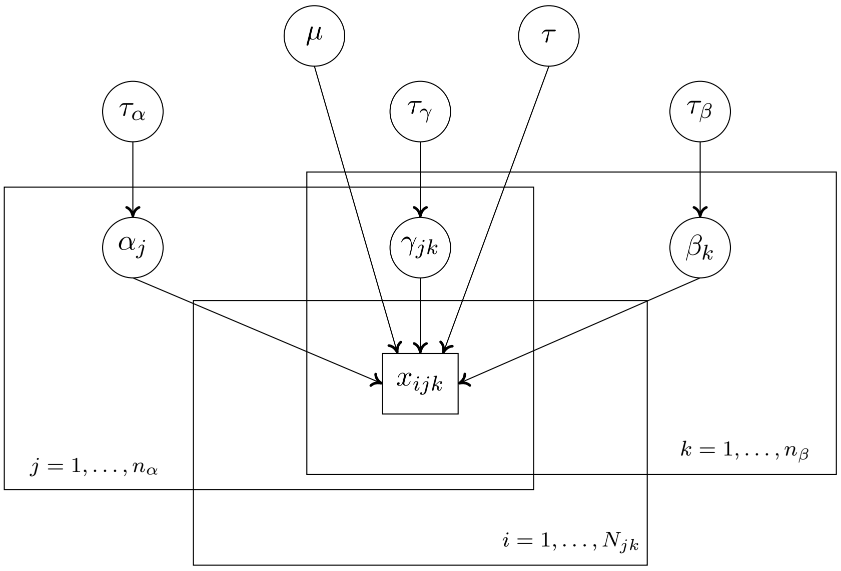


Figure 3.1: DAG for the saturated model

The prior distributions, the likelihood of the model and the derivation of the full conditionals of the unknown parameters are formed as follows:

Priors

- For $\tau \sim \Gamma(a, b)$

$$f(\tau|a,b) = \frac{b^a \tau^{a-1} e^{-b\tau}}{\Gamma(a)} \propto \tau^{a-1} e^{-b\tau}.$$

- For $\mu \sim$ N(μ_0,τ_0^{-1})

$$f(\mu|\mu_0, \tau_0) = \sqrt{\frac{\tau_0}{2\pi}} e^{\frac{-\tau_0(\mu-\mu_0)^2}{2}} \propto \sqrt{\tau_0} e^{\frac{-\tau_0(\mu-\mu_0)^2}{2}} = \tau_0^{\frac{1}{2}} e^{-\frac{\tau_0}{2}(\mu-\mu_0)^2}.$$

- For $\alpha_j \sim$ N(μ_α,τ_α^{-1}) [The effects of α]

Similarly,

$$f(\alpha_j|\mu_\alpha, \tau_\alpha) \propto \sqrt{\tau_\alpha} e^{\frac{-\tau_\alpha(\alpha_j-\mu_\alpha)^2}{2}} = \tau_\alpha^{\frac{1}{2}} e^{-\frac{\tau_\alpha}{2}(\alpha_j-\mu_\alpha)^2}.$$

So,

$$f(\alpha|\mu_\alpha,\tau_\alpha)\propto\tau_\alpha^{\frac{n_\alpha}{2}}e^{-\frac{\tau_\alpha}{2}\sum\limits_{j=1}^{n_\alpha}(\alpha_j-\mu_\alpha)^2}.$$

- For β_k ~ N(μ_β,τ_β^{-1}) [The effects of β]

Similarly,

$$f(\beta_k|\mu_\beta,\tau_\beta)\propto\sqrt{\tau_\beta}e^{\frac{-\tau_\beta(\beta_k-\mu_\beta)^2}{2}}=\tau_\beta^{\frac{1}{2}}e^{-\frac{\tau_\beta}{2}(\beta_k-\mu_\beta)^2}.$$

So,

$$f(\beta|\mu_\beta,\tau_\beta)\propto\tau_\beta^{\frac{n_\beta}{2}}e^{-\frac{\tau_\beta}{2}\sum\limits_{j=1}^{n_\beta}(\beta_k-\mu_\beta)^2}.$$

- For γ_{jk} ~ N(μ_γ,τ_γ^{-1}) [The effects of interactions γ]

Similarly,

$$f(\gamma_{jk}|\mu_\gamma,\tau_\gamma)\propto\sqrt{\tau_\gamma}e^{\frac{-\tau_\gamma(\gamma_{jk}-\mu_\gamma)^2}{2}}=\tau_\gamma^{\frac{1}{2}}e^{-\frac{\tau_\gamma}{2}(\gamma_{jk}-\mu_\gamma)^2}.$$

So,

$$f(\gamma|\mu_\gamma,\tau_\gamma)\propto\tau_\gamma^{\frac{n_\gamma}{2}}e^{-\frac{\tau_\gamma}{2}\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}(\gamma_{jk}-\mu_\gamma)^2}.$$

- For τ_α ~ $\Gamma(a_{\tau_\alpha},b_{\tau_\alpha})$

$$f(\tau_\alpha|a_{\tau_\alpha},b_{\tau_\alpha})=\frac{b^{a_{\tau_\alpha}}\tau_\alpha^{a_{\tau_\alpha}-1}e^{-b_{\tau_\alpha}\tau_\alpha}}{\Gamma(a)}\propto\tau_\alpha^{a_{\tau_\alpha}-1}e^{-b_{\tau_\alpha}\tau_\alpha}.$$

- For τ_β ~ $\Gamma(a_{\tau_\beta},b_{\tau_\beta})$

Similarly,

$$f(\tau_\beta|a_{\tau_\beta},b_{\tau_\beta})\propto\tau_\beta^{a_{\tau_\beta}-1}e^{-b_{\tau_\beta}\tau_\beta}.$$

- For τ_γ ~ $\Gamma(a_{\tau_\gamma},b_{\tau_\gamma})$

Similarly,

$$f(\tau_\gamma|a_{\tau_\gamma},b_{\tau_\gamma})\propto\tau_\gamma^{a_{\tau_\gamma}-1}e^{-b_{\tau_\gamma}\tau_\gamma}.$$

Likelihood

$$\begin{aligned}p(\underline{\underline{x}}|\tau,\mu,\underline{\alpha},\underline{\beta},\underline{\underline{\gamma}})&=\prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\prod_{i=1}^{N_{jk}}\left(\sqrt{\frac{\tau}{2\pi}}e^{\frac{-\tau\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2}{2}}\right)\\&=(2\pi)^{-\frac{N}{2}}\tau^{\frac{N}{2}}e^{-\frac{\tau}{2}\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}\sum\limits_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2}.\end{aligned}$$

Joint density

The joint density is the likelihood multiplied by the priors.

$$
\begin{aligned}
p(\underline{\underline{x}}, \tau, \mu, \underline{\alpha}, \underline{\beta}, \underline{\underline{\gamma}}) = {} & \underbrace{(2\pi)^{-\frac{N}{2}} \tau^{\frac{N}{2}} e^{-\frac{\tau}{2}\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}\sum\limits_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2}}_{\text{Likelihood}} \underbrace{\tau^{a-1}e^{-b\tau}}_{\text{prior } \tau} \\
& \underbrace{\tau_0^{\frac{1}{2}} e^{-\frac{\tau_0}{2}(\mu-\mu_0)^2}}_{\text{prior } \mu} \underbrace{\tau_\alpha^{\frac{n_\alpha}{2}} e^{-\frac{\tau_\alpha}{2}\sum\limits_{j=1}^{n_\alpha}(\alpha_j-\mu_\alpha)^2}}_{\text{prior } \alpha\text{s}} \underbrace{\tau_\beta^{\frac{n_\beta}{2}} e^{-\frac{\tau_\beta}{2}\sum\limits_{j=1}^{n_\beta}(\beta_k-\mu_\beta)^2}}_{\text{prior } \beta\text{s}} \\
& \underbrace{\tau_\gamma^{\frac{n_\gamma}{2}} e^{-\frac{\tau_\gamma}{2}\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}(\gamma_{jk}-\mu_\gamma)^2}}_{\text{prior } \gamma\text{s}} \underbrace{\tau_\alpha^{a_{\tau_\alpha}-1}e^{-b_{\tau_\alpha}\tau_\alpha}}_{\text{prior } \tau_\alpha} \underbrace{\tau_\beta^{a_{\tau_\beta}-1}e^{-b_{\tau_\beta}\tau_\beta}}_{\text{prior } \tau_\beta} \\
& \underbrace{\tau_\gamma^{\alpha_{\tau_\gamma}-1}e^{-\beta_{\tau_\gamma}\tau_\gamma}}_{\text{prior } \tau_\gamma}.
\end{aligned}
$$

After extracting the common factors and using *exp* for the exponential for clarity, the joint density for the saturated model becomes:

$$
\begin{aligned}
p(\underline{\underline{x}}, \tau, \mu, \underline{\alpha}, \underline{\beta}, \underline{\underline{\gamma}}) = (2\pi)^{-\frac{N}{2}} \exp\Bigg\{ & -\tau\Bigg[b + \frac{1}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk})^2\Bigg] \\
& - \frac{\tau_0}{2}(\mu-\mu_0)^2 - \frac{\tau_\alpha}{2}\sum_{j=1}^{n_\alpha}(\alpha_j-\mu_\alpha)^2 - \frac{\tau_\beta}{2}\sum_{j=1}^{n_\beta}(\beta_k-\mu_\beta)^2 \\
& - \frac{\tau_\gamma}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}(\gamma_{jk}-\mu_\gamma)^2 - \beta_{\tau_\alpha}\tau_\alpha - \beta_{\tau_\beta}\tau_\beta - \beta_{\tau_\gamma}\tau_\gamma\Bigg\}\tau^{a+\frac{N}{2}-1} \\
& \tau_\alpha^{a_{\tau_\alpha}+\frac{n_\alpha}{2}-1}\tau_\beta^{a_{\tau_\beta}+\frac{n_\beta}{2}-1}\tau_\gamma^{a_{\tau_\gamma}+\frac{n_\gamma}{2}-1}.
\end{aligned}
$$

Full conditionals

For the posterior probability densities we adopt the letter π, that is often used in the literature to denote the target distribution. Furthermore, when displaying the full conditional distribution of a variable, conditioning on "all other variables" is denoted using a dot e.g. `variable_of_interest`$|\cdot$.

- For τ

$$
\pi(\tau|\cdot) \propto \tau^{a+\frac{N}{2}-1}e^{-\tau\left(b+\frac{1}{2}\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}\sum\limits_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2\right)}.
$$

So,

$$\tau|\cdot \sim \; Ga\left(a+\frac{N}{2}, b+\frac{1}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2\right).$$

- For α_j

$$\begin{aligned}
\pi(\alpha_j|\cdot) &\propto \exp\left\{-\frac{1}{2}\left((\alpha_j-\mu_\alpha)^2\tau_\alpha+\tau\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2\right)\right\} \\
&\propto \exp\Bigg\{-\frac{1}{2}\Bigg((\alpha_j-\mu_\alpha)^2\tau_\alpha+\tau\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}(\alpha_j^2-2x_{ijk}\alpha_j+2\mu\alpha_j+2\beta_k\alpha_j \\
&\quad +2\alpha_j\gamma jk)\Bigg)\Bigg\} \\
&\propto \exp\Bigg\{-\frac{1}{2}\alpha_j^2\tau_\alpha+\alpha_j\mu_\alpha\tau_\alpha-\frac{1}{2}\tau N_j\alpha_j^2+\tau\alpha_j\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}x_{ijk}-\tau N_j\mu\alpha_j \\
&\quad -\tau\alpha_j\sum_{k=1}^{n_\beta}\beta_k N_{jk}-\tau\alpha_j\sum_{k=1}^{n_\beta}\gamma_{jk}N_{jk}\Bigg\} \\
&\propto \exp\Bigg\{\alpha_j^2\left(-\frac{1}{2}\tau_\alpha-\frac{1}{2}\tau N_j\right)+\alpha_j\Bigg(\mu_\alpha\tau_\alpha+\tau\Bigg(\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}x_{ijk}-N_j\mu-\sum_{k=1}^{n_\beta}\beta_k N_{jk} \\
&\quad -\sum_{k=1}^{n_\beta}\gamma_{jk}N_{jk}\Bigg)\Bigg)\Bigg\}.
\end{aligned}$$

So,

$$\alpha_j|\cdot \sim \; N\left(\frac{\mu_\alpha\tau_\alpha+\tau\left(\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}x_{ijk}-N_j\mu-\sum_{k=1}^{n_\beta}\beta_k N_{jk}-\sum_{k=1}^{n_\beta}\gamma_{jk}N_{jk}\right)}{\tau_\alpha+\tau N_j}, \frac{1}{\tau_\alpha+\tau N_j}\right).$$

- For β_k

Similarly with the full conditional of α_j.
So,

$$\beta_k|\cdot \sim \; N\left(\frac{\mu_\beta\tau_\beta+\tau\left(\sum_{j=1}^{n_\alpha}\sum_{i=1}^{N_{jk}}x_{ijk}-N_k\mu-\sum_{j=1}^{n_\alpha}\alpha_j N_{jk}-\sum_{j=1}^{n_\alpha}\gamma_{jk}N_{jk}\right)}{\tau_\beta+\tau N_k}, \frac{1}{\tau_\beta+\tau N_k}\right).$$

- For μ

Similarly with the full conditional of α_j.

$$\mu|\cdot \sim N\left(\frac{\mu_0\tau_0 + \tau\left(\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}\sum\limits_{i=1}^{N_{jk}} x_{ijk} - \sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}\alpha_j N_{jk} - \sum\limits_{k=1}^{n_\beta}\sum\limits_{j=1}^{n_\alpha}\beta_k N_{jk} - \sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta}\gamma_{jk}N_{jk}\right)}{\tau_0+\tau N}, \frac{1}{\tau_0+\tau N}\right).$$

- For γ_{jk}

$$\begin{aligned}
\pi(\gamma_{jk}|\cdot) &\propto \exp\left\{-\frac{1}{2}\left[(\gamma-\mu_\gamma)^2\tau_\gamma + \tau\sum_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2\right]\right\} \\
&\propto \exp\left\{-\frac{1}{2}\left[(\gamma-\mu_\gamma)^2\tau_\gamma + \tau\left(-2\sum_{i=1}^{N_{kj}} x_{ijk}\gamma + 2N_{jk}\mu_\gamma + 2\beta_k N_{jk}\gamma\right.\right.\right. \\
&\quad \left.\left.\left. +2\alpha_j N_{jk}\gamma + N_{jk}\gamma\right)^2\right]\right\} \\
&\propto \exp\left\{\gamma^2\left(-\frac{1}{2}\tau_\gamma - \frac{1}{2}\tau N_{jk}\right) + \gamma\left[\mu_\gamma\tau_\gamma + \tau\left(\sum_{i=1}^{N_{jk}} x_{ijk} + N_{jk}(-\mu-\alpha_j-\beta_k)\right)\right]\right\}
\end{aligned}$$

So,

$$\gamma_{jk}|\cdot \sim N\left(\frac{\mu_\gamma\tau_\gamma + \tau\left(\sum\limits_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu+\alpha_j+\beta_k)\right)}{\tau_\gamma+\tau N_{jk}}, \frac{1}{\tau_\gamma+\tau N_{jk}}\right).$$

- For τ_α

$$\begin{aligned}
\pi(\tau_\alpha|\cdot) &\propto \tau_\alpha^{a_{\tau_\alpha}-1}e^{-b_{\tau_\alpha}\tau_\alpha}\prod_{j=1}^{n_\alpha}\left(\sqrt{\tau_{\alpha_j}}e^{\frac{-\tau_\alpha(\alpha_j-\mu_\alpha)^2}{2}}\right) \\
&\propto \tau_\alpha^{a_{\tau_\alpha}-1}e^{-b_{\tau_\alpha}\tau_\alpha}\tau_a^{\frac{n_\alpha}{2}}e^{\sum\limits_{j=1}^{n_\alpha}\frac{-\tau_a(\alpha_j-\mu_\alpha)^2}{2}} \\
&\propto \tau_\alpha^{a_{\tau_\alpha}+\frac{n_\alpha}{2}-1}e^{-\tau_\alpha\left(b_{\tau_\alpha}+\frac{1}{2}\sum\limits_{j=1}^{n_\alpha}(\alpha_j-\mu_\alpha)^2\right)}.
\end{aligned}$$

So,

$$\tau_\alpha|\cdot \sim \; \Gamma\left(\alpha_{\tau_\alpha} + \frac{n_\alpha}{2}, \beta_{\tau_\alpha} + \frac{1}{2}\sum_{j=1}^{n_\alpha}(\alpha_j - \mu_\alpha)^2\right).$$

- For τ_β

Similarly,

$$\pi(\tau_\beta|\cdot) \propto \tau_\beta^{a_{\tau_\beta}-1} e^{-b_{\tau_\beta}\tau_\beta} \prod_{k=1}^{n_\beta}\left(\sqrt{\tau_\beta} e^{\frac{-\tau_\beta(\beta_k-\mu_\beta)^2}{2}}\right).$$

So,

$$\tau_\beta|\cdot \sim \; \Gamma\left(a_{\tau_\beta} + \frac{n_\beta}{2}, b_{\tau_\beta} + \frac{1}{2}\sum_{k=1}^{n_\beta}(\beta_k - \mu_\beta)^2\right).$$

- For τ_γ

Similarly,

$$\pi(\tau_\gamma|\cdot) \propto \tau_\gamma^{a_{\tau_\gamma}-1} e^{-b_{\tau_\gamma}\tau_\gamma} \prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\left(\sqrt{\tau_\gamma} e^{\frac{-\tau_\gamma(\gamma_{jk}-\mu_\gamma)^2}{2}}\right).$$

So,

$$\tau_\gamma|\cdot \sim \; \Gamma\left(a_{\tau_\gamma} + \frac{n_\gamma}{2}, b_{\tau_\gamma} + \frac{1}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}(\gamma_{jk} - \mu_\gamma)^2\right).$$

3.3.2 Variable selection using variable selection indicators

The saturated model of the previous section (3.3.1), depending on the number of levels of the categorical variables, may involve numerous parameters that increase the complexity of the model and reduce its interpretability. Dimensionality reduction can be applied in order to tackle this problem.

The first technique, by which we will implement Bayesian variable selection, is the Kuo and Mallick approach (Kuo and Mallick, 1998). We saw in Section 2.4.1 (1a) that this method requires the use of a variable selection indicator I, to indicate the presence or absence of a particular covariate in the model. The indicators follow a Bernoulli distribution with inclusion probability p. Sometimes p is fixed. However, introducing a Beta prior on the inclusion probability allows the model to learn about the degree of sparsity from the data. Here, we are embedding this prior in the model. An auxiliary variable denoted as θ is also added to represent the effect size of the interaction covariate γ_{jk}, so that $\gamma_{jk} = I_{jk} \cdot \theta_{jk}$.

In the following sections, we will examine all four variations of symmetry for our two-way Anova model, which includes variable selection via the Kuo and Mallick approach. In the first case, where both the main and interaction effects are asymmetric, we will declare all the prior distributions. In the following cases, to avoid repetition, we will only present analytically the prior distributions that change. In addition, all main and interaction effects are treated as random, hence, we consider that the mean of their prior distribution is zero and a gamma prior is assigned to their precisions.

3.3.2.1 Asymmetric main and asymmetric interaction effects

In this case, the two main effects come from different distributions. We do not assume any symmetry information for the interactions effects either. The model has a lot of common elements with the saturated case 3.3.1 and is defined as:

Model:

$X_{ijk}|\mu, \alpha_j, \beta_k, I_{jk}, \theta_{jk}, \tau \sim \text{N}(\mu + \alpha_j + \beta_k + I_{jk}\theta_{jk},\ \tau^{-1})$, i=1,...,$N_{jk}$,

$N = \sum\limits_{j=1}^{n_\alpha} \sum\limits_{k=1}^{n_\beta} N_{jk}$

$\mu \sim \text{N}(\mu_0,\tau_0^{-1})$

$\tau \sim \mathit{\Gamma}(a, b)$

$\alpha_j|\tau_\alpha \sim \text{N}(\mu_\alpha,\tau_\alpha^{-1})$, j=1,...,$n_\alpha$

$\beta_k|\tau_\beta \sim \text{N}(\mu_\beta,\tau_\beta^{-1})$, k=1,...,$n_\beta$

$\theta_{jk}|\tau_\theta \sim \text{N}(\mu_\theta,\tau_\theta^{-1})$

$I_{jk}|p \sim \mathit{Bern}(p)$

$p \sim \mathit{Beta}(a_p, b_p)$

$\tau_\alpha \sim \mathit{\Gamma}(a_{\tau_\alpha},b_{\tau_\alpha})$

$\tau_\beta \sim \mathit{\Gamma}(a_{\tau_\beta},b_{\tau_\beta})$

$\tau_\theta \sim \mathit{\Gamma}(a_{\tau_\theta},b_{\tau_\theta})$

The corresponding DAG for the model is presented in Figure 3.2. It includes a deterministic node (marked in a double circle) for the product of each variable selection indicator and the corresponding effect size.

The prior distributions, the likelihood of the model and the derivation of the full conditionals of the unknown parameters are formed as follows:

Priors

To the priors mentioned in Section 3.3.1 we add the priors for the indicator variable and the inclusion probability. Furthermore, priors for θ_{jk} and τ_θ replace the priors for γ_{jk} and τ_γ.

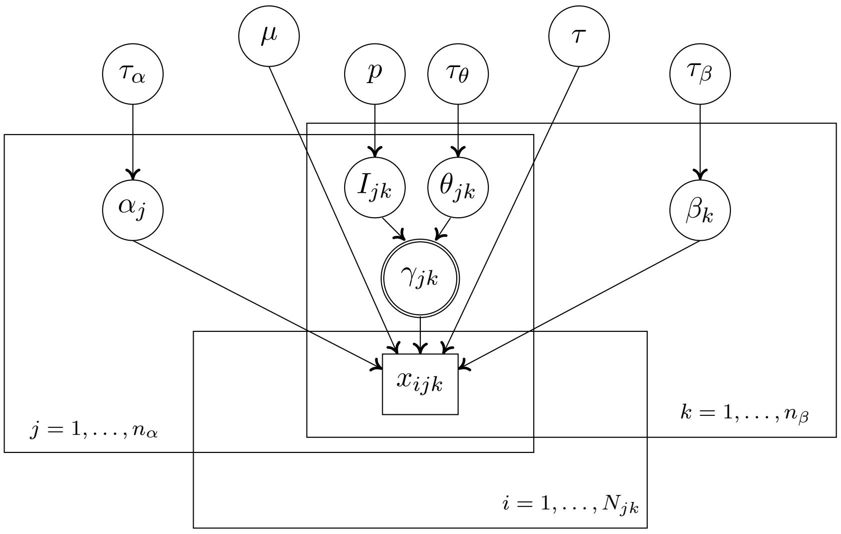


Figure 3.2: DAG for variable selection for asymmetric main and interaction effects

- $I_{jk}|p \sim Bern(p)$

$$f(I_{jk}|p) = p^{I_{jk}}(1-p)^{1-I_{jk}}.$$

- $p \sim Beta(a_p, b_p)$

$$\begin{aligned} f(p|a_p, b_p) &= \frac{p^{a_p-1}(1-p)^{b_p-1}}{B(a_p, b_p)} \\ &\propto p^{a_p-1}(1-p)^{b_p-1}. \end{aligned}$$

Likelihood

The likelihood here is similar to the likelihood of the saturated model. Now instead of γs there are Is and θs.

$$\begin{aligned} p(\underline{\underline{x}}|\tau s, \mu, p, \underline{\alpha}, \underline{\beta}, \underline{\underline{I}}, \underline{\underline{\theta}}) &= \prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\prod_{i=1}^{N_{jk}}\left(\sqrt{\frac{\tau}{2\pi}}e^{\frac{-\tau\left(x_{ijk}-\mu-\alpha_j-\beta_k-I_{jk}\theta_{jk}\right)^2}{2}}\right) \\ &= (2\pi)^{-\frac{N}{2}}\tau^{\frac{N}{2}}e^{-\frac{\tau}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-I_{jk}\theta_{jk}\right)^2}. \end{aligned}$$

Joint density

The joint density after multiplying the likelihood by the priors and simplifying becomes:

$$
\begin{aligned}
p(\underline{\underline{x}}, \tau, \mu, \underline{\alpha}, \underline{\beta}, \underline{\underline{I}}, \underline{\underline{\theta}}) = (2\pi)^{-\frac{N}{2}} \exp\Bigg\{ & -\tau\Bigg[b + \frac{1}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}(x_{ijk} - \mu - \alpha_j - \beta_k - I_{jk}\theta_{jk})^2\Bigg] \\
& - \frac{\tau_0}{2}(\mu - \mu_0)^2 - \frac{\tau_\alpha}{2}\sum_{j=1}^{n_\alpha}(\alpha_j - \mu_\alpha)^2 - \frac{\tau_\beta}{2}\sum_{j=1}^{n_\beta}(\beta_k - \mu_\beta)^2 \\
& - \frac{\tau_\theta}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}(I_{jk}\theta_{jk} - \mu_\theta)^2 - \beta_{\tau_\alpha}\tau_\alpha - \beta_{\tau_\beta}\tau_\beta - \beta_{\tau_\theta}\tau_\theta\Bigg\}\tau^{a+\frac{N}{2}-1} \\
& \tau_\alpha^{a_{\tau_\alpha}+\frac{n_\alpha}{2}-1}\tau_\beta^{a_{\tau_\beta}+\frac{n_\beta}{2}-1}\tau_\theta^{a_{\tau_\theta}+\frac{n_\gamma}{2}-1} p^{\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta} I_{jk}+a_p-1}(1-p)^{n_\gamma-\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta} I_{jk}+b_p-1}.
\end{aligned}
$$

Full conditionals

The full conditionals for τ, μ, α_j and β_k change to include the variable selection indicators and are estimated in the same way as in the previous section for the saturated model (3.3.1). We only illustrate how τ changes to incorporate I_{jk} and θ_{jk}, with the other variables following the same logic of replacing γ with the product of I and θ. Precisions for the main effects, τ_α and τ_β, remain the same as for the saturated model. Full conditionals for τ_γ, γ_{jk} are replaced with full conditionals for τ_θ, θ_{jk}, estimated in the same way. Additionally, we calculate the full conditionals for I_{jk} and p.

- For τ

$$
\tau|\cdot \sim\ Ga\left(a + \frac{N}{2}, b + \frac{1}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}(x_{ijk} - \mu - \alpha_j - \beta_k - I_{jk}\theta_{jk})^2\right).
$$

- For τ_θ

$$
\pi(\tau_\theta|\cdot) \propto \tau_\theta^{a_{\tau_\theta}-1}e^{-b_{\tau_\theta}\tau_\theta}\prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\left(\sqrt{\tau_\theta}e^{\frac{-\tau_\theta(\theta_{jk}-\mu_\theta)^2}{2}}\right).
$$

So,

$$
\tau_\theta|\cdot \sim\ \Gamma\left(a_{\tau_\theta} + \frac{n_\gamma}{2}, b_{\tau_\theta} + \frac{1}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}(\theta_{jk} - \mu_\theta)^2\right).
$$

- For θ_{jk}

There are two cases; if I_{jk}=0 and if I_{jk}=1.

- If I_{jk}=0,

$$\theta_{jk}|I_{jk}=0,\cdot \sim \; N\left(\mu_\theta, \tau_\theta^{-1}\right),$$

which is simply the prior.

- If I_{jk}=1,

$$\theta_{jk}|I_{jk}=1,\cdot \sim \; N\left(\frac{\mu_\theta\tau_\theta+\tau\left(\sum_{i=1}^{N_{jk}} x_{ijk}-N_{jk}(\mu+\alpha_j+\beta_k)\right)}{\tau_\theta+\tau N_{jk}}, \frac{1}{\tau_\theta+\tau N_{jk}}\right).$$

- For I_{jk}

According to the Law of Total Probability

$$\pi(I|y)=\frac{\pi(I,y)}{\pi(0,y)+\pi(1,y)}.$$

We want the probability:

$$Pr(I_{jk}=1|\cdot)=\frac{\pi(I_{jk}=1|\cdot)}{\pi(I_{jk}=1|\cdot)+\pi(I_{jk}=0|\cdot)},$$

and we know that:

$$\begin{aligned}
\pi(I_{jk}|\cdot) &\propto p^{I_{jk}}(1-p)^{1-I_{jk}}\exp\left\{-\frac{\tau}{2}\sum_{i=1}^{N_{jk}}(x_{ijk}-\mu-\alpha_j-\beta_k-I_{jk}\theta_{jk})^2\right\}\\
&\propto p^{I_{jk}}(1-p)^{1-I_{jk}}\exp\left\{\tau\sum_{i=1}^{N_{jk}}\left(x_{ijk}I_{jk}\theta_{jk}-\mu I_{jk}\theta_{jk}-\alpha_j I_{jk}\theta_{jk}-\beta_j I_{jk}\theta_{jk}\right.\right.\\
&\quad\left.\left.-\frac{1}{2}I_{jk}\theta_{jk}^2\right)\right\}\\
&\propto p^{I_{jk}}(1-p)^{1-I_{jk}}\exp\left\{\tau\left[I_{jk}\theta_{jk}\sum_{i=1}^{N_{jk}}x_{ijk}-N_{jk}\left(\mu I_{jk}\theta_{jk}+\alpha_j I_{jk}\theta_{jk}+\beta_k I_{jk}\theta_{jk}\right.\right.\right.\\
&\quad\left.\left.\left.+\frac{1}{2}I_{jk}\theta_{jk}^2\right)\right]\right\}\\
&\propto p^{I_{jk}}(1-p)^{1-I_{jk}}\exp\left\{\tau I_{jk}\theta_{jk}\left[\sum_{i=1}^{N_{jk}}x_{ijk}-N_{jk}\left(\mu+\alpha_j+\beta_k+\frac{1}{2}\theta_{jk}\right)\right]\right\}.
\end{aligned}$$

Then we normalise,

$$Pr(I=0|\cdot) = \frac{1-p}{1-p+p\exp\left\{\tau\theta_{jk}\left(\sum\limits_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu+\alpha_j+\beta_k+\frac{1}{2}\theta_{jk})\right)\right\}}$$

$$Pr(I=1|\cdot) = \frac{p\exp\left\{\tau\theta_{jk}\left(\sum\limits_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu+\alpha_j+\beta_k+\frac{1}{2}\theta_{jk})\right)\right\}}{1-p+p\exp\left\{\tau\theta_{jk}\left(\sum\limits_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu+\alpha_j+\beta_k+\frac{1}{2}\theta_{jk})\right)\right\}}.$$

- For p

$$\begin{aligned}\pi(p|\cdot) &\propto p^{a_p-1}(1-p)^{b_p-1}\prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\left(p^{I_{jk}}(1-p)^{1-I_{jk}}\right)\\ &\propto p^{a_p-1}(1-p)^{b_p-1}p^{\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta} I_{jk}}(1-p)^{n_\gamma-\sum\limits_{j=1}^{n_\alpha}\sum\limits_{k=1}^{n_\beta} I_{jk}}.\end{aligned}$$

So,

$$p|\cdot \sim\ B\left(a_p+\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta} I_{jk}, b_p+n_\gamma+\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta} I_{jk}\right).$$

3.3.2.2 Symmetric main and asymmetric interaction effects

For the case where the main effects are treated as symmetric and the interaction effects as asymmetric, we assume that both main effects are the same. There is only one set denoted z, with n_z number of levels, z_j, $j=1,...,n_z$. Every z_j is updated at every step of the Gibbs sampler by its full conditional distribution that takes into consideration all the observations for which either the first or the second independent variable is equal to j, or they both are.

The model for this case is defined as follows:

Model:

$X_{ijk}|\mu, z_j, z_k, I_{jk}, \theta_{jk}, \tau \sim \text{N}(\mu + z_j + z_k + I_{jk}\theta_{jk},\ \tau^{-1})$, i=1,...,$N_{jk}$,

$N = \sum\limits_{j=1}^{n_\alpha} \sum\limits_{k=1}^{n_\beta} N_{jk}$

$\mu \sim \text{N}(\mu_0, \tau_0^{-1})$

$\tau \sim \Gamma(a, b)$

$z_j|\tau_z \sim \text{N}(0, \tau_z^{-1})$, j=1,...,$n_z$

$\theta_{jk}|\tau_\theta \sim \text{N}(\mu_\theta, \tau_\theta^{-1})$

$I_{jk} \sim Bern(p)$

$p \sim Beta(a_p, b_p)$

$\tau_z \sim \Gamma(a_{\tau_z}, b_{\tau_z})$

$\tau_\theta \sim \Gamma(a_{\tau_\theta}, b_{\tau_\theta})$

The notation used for this model changes slightly from the previous cases and is the following:

Notation:

n_z is the number of levels of the independent categorical variable z

N_{jk} is the number of observations for which the first independent variable is z_j and the second is z_k and $j \neq k$

N_{kj} is the number of observations for which the first independent variable is z_k and the second is z_j and $j \neq k$

N_j is the number of observations for which either the first or the second independent variable is z_j, but not both

N_{jj} is the number of observations for which both zs are j

The corresponding DAG for the model including variable selection and symmetric main effects is presented in Figure 3.3. It includes one level of main effects indexed twice to indicate that it appears twice in the model.

There is a new prior added for the symmetric main effects z, similar to the prior distributions that we had for α and β. The likelihood and the posterior distributions change to incorporate this new concept.

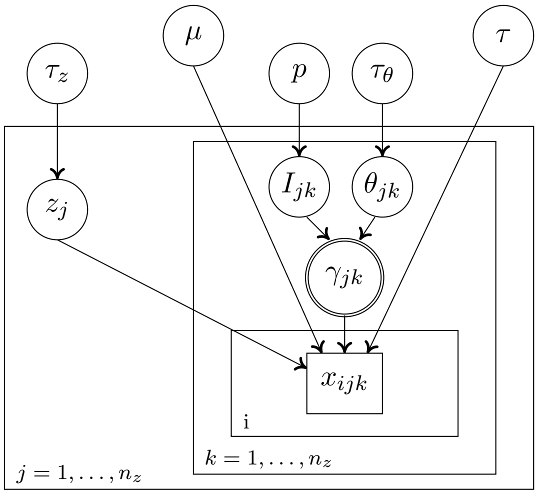


Figure 3.3: DAG for variable selection for symmetric main effects

Priors

- $z_j \sim \mathrm{N}(\mu_z, \tau_z^{-1})$

$$f(z_j|\mu_z, \tau_z) \propto \sqrt{\tau_z} e^{\frac{-\tau_z(z_j-\mu_z)^2}{2}} = \tau_z^{\frac{1}{2}} e^{-\frac{\tau_z}{2}(z_j-\mu_z)^2}.$$

So,

$$f(z|\mu_z, \tau_z) \propto \tau_z^{\frac{n_\alpha}{2}} e^{-\frac{\tau_z}{2}\sum\limits_{j=1}^{n_\alpha}(z_j-\mu_z)^2}.$$

Likelihood

The likelihood changes to:

$$\begin{aligned} p(\underline{x}|\tau, \mu, \underline{z}, \underline{I}, \underline{\theta}) &= \prod_{j=1}^{n_z}\prod_{k=1}^{n_z}\prod_{i=1}^{N_{jk}} \left(\sqrt{\frac{\tau}{2\pi}} e^{\frac{-\tau\left(x_{ijk}-\mu-z_j-z_k-I_{jk}\theta_{jk}\right)^2}{2}}\right) \\ &= (2\pi)^{-\frac{N}{2}} \tau^{\frac{N}{2}} e^{-\frac{\tau}{2}\sum\limits_{j=1}^{n_z}\sum\limits_{k=1}^{n_z}\sum\limits_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-z_j-z_k-I_{jk}\theta_{jk}\right)^2}. \end{aligned}$$

Joint density

The joint density is the result of multiplying all the prior distributions with the likelihood. Since we have not repeated all the priors in this section, and the logic of the model has changed, for clarity we are presenting the joint density explicitly without factoring out the common terms.

$$
\begin{aligned}
p(\underline{\underline{x}}, \tau, \mu, \underline{\alpha}, \underline{\beta}, \underline{\underline{\gamma}}) = {} & \underbrace{(2\pi)^{-\frac{N}{2}} \tau^{\frac{N}{2}} e^{-\frac{\tau}{2} \sum\limits_{j=1}^{n_z} \sum\limits_{k=1}^{n_z} \sum\limits_{i=1}^{N_{jk}} \left(x_{ijk}-\mu-z_j-z_k-I_{jk}\theta_{jk}\right)^2}}_{\text{Likelihood}} \underbrace{\tau^{a-1} e^{-b\tau}}_{\text{prior } \tau} \\
& \underbrace{\tau_0^{\frac{1}{2}} e^{-\frac{\tau_0}{2}(\mu-\mu_0)^2}}_{\text{prior } \mu} \underbrace{\tau_z^{\frac{n_z}{2}} e^{-\frac{\tau_z}{2} \sum\limits_{j=1}^{n_z} (z_j-\mu_z)^2}}_{\text{prior } z\text{s}} \underbrace{\tau_\theta^{\frac{n_\gamma}{2}} e^{-\frac{\tau_\theta}{2} \sum\limits_{j=1}^{n_z} \sum\limits_{k=1}^{n_z} (\theta_{jk}-\mu_\theta)^2}}_{\text{prior } \theta\text{s}} \\
& \underbrace{\tau_z^{a_{\tau_z}-1} e^{-b_{\tau_z}\tau_z}}_{\text{prior } \tau_z} \underbrace{\tau_\theta^{a_{\tau_\theta}-1} e^{-b_{\tau_\theta}\tau_\theta}}_{\text{prior } \tau_\theta} \underbrace{p^{a_p-1}(1-p)^{b_p-1}}_{\text{prior } p} \\
& \underbrace{p^{\sum\limits_{j=1}^{n_z} \sum\limits_{k=1}^{n_z} I_{jk}} (1-p)^{n_\gamma - \sum\limits_{j=1}^{n_z} \sum\limits_{k=1}^{n_z} I_{jk}}}_{\text{prior } I\text{s}}.
\end{aligned}
$$

Full conditionals

The full conditionals change equivalently to incorporate the notion of the symmetric main effects.

- For τ

$$
\tau|\cdot \sim \; Ga\left(a + \frac{N}{2}, b + \frac{1}{2} \sum_{j=1}^{n_z} \sum_{k=1}^{n_z} \sum_{i=1}^{N_{jk}} \left(x_{ijk} - \mu - z_j - z_k - I_{jk}\theta_{jk}\right)^2\right).
$$

- For μ

$$
\mu|\cdot \sim \; N\left(\frac{\mu_0\tau_0 + \tau\left(\sum\limits_{j=1}^{n_z} \sum\limits_{k=1}^{n_z} \sum\limits_{i=1}^{N_{jk}} x_{ijk} - \sum\limits_{j=1}^{n_l} z_j N_j - \sum\limits_{k=1}^{n_l} z_k N_k - \sum\limits_{j=1}^{n_z} \sum\limits_{k=1}^{n_z} I_{jk}\theta_{jk}N_{jk}\right)}{\tau_0 + \tau N}, \right.
$$

$$
\left. \frac{1}{\tau_0 + \tau N}\right).
$$

- For z_j

$$\pi(z_j|\cdot) \propto \exp\left\{-\frac{1}{2}\left((z_j-\mu_z)^2\tau_z + \tau\sum_{k=1}^{n_z}\sum_{i=1}^{N_{jk}}(x_{ijk}-\mu-z_j-z_k-I_{jk}\theta_{jk})^2\right)\right\}$$

$$\propto \exp\left\{-\frac{1}{2}\left[(z_j-\mu_z)^2\tau_z + \tau\left(\overbrace{\sum_{\substack{k=1\\k\neq l}}^{n_z}\sum_{i=1}^{N_{jk}}\left(z_j^2-2x_{ijk}z_j-2x_{ikj}z_j+2\mu z_j+2z_kz_j+2z_jI_{jk}\theta jk\right)^2}^{\substack{\text{for the observations with}\\ z_j\text{ on either side, not both}}}\right.\right.\right.$$

$$\left.\left.\left. + \overbrace{\sum_{i=1}^{N_{jj}}\left(4z_j^2-4x_{ijj}z_j+4\mu z_j+4z_jI_{jj}\theta jj\right)^2}^{\substack{\text{for the observations with}\\ z_j\text{ on both sides}}}\right)\right]\right\}$$

$$\propto \exp\left\{-\frac{1}{2}\left[(z_j-\mu_z)^2\tau_z + \tau\left(N_jz_j^2 - 2z_j\sum_{\substack{k=1\\k\neq j}}^{n_z}\sum_{i=1}^{N_{jk}}x_{ijk} - 2z_j\sum_{\substack{k=1\\k\neq j}}^{n_z}\sum_{i=1}^{N_{jk}}x_{ikj} + 2z_j\sum_{\substack{k=1\\k\neq l}}^{n_z}z_kN_j\right.\right.\right.$$

$$+ 2z_j\sum_{\substack{k=1\\k\neq l}}^{n_z}I_{jk}\theta_{jk}N_{jk} + 2z_j\sum_{\substack{k=1\\k\neq l}}^{n_z}I_{jk}\theta_{kj}N_{kj} + 2N_j\mu z_j + 4z_j^2N_{jj} - 4z_j\sum_{i=1}^{N_{jj}}x_{ijj}$$

$$\left.\left.\left. + 4N_{jj}\mu z_j + 4z_jI_{jj}\theta_{jj}N_{jj}\right)\right]\right\}$$

$$\propto \exp\left\{z_j^2\left(-\frac{1}{2}\tau_z-\frac{1}{2}\tau N_j-2N_{jj}\tau\right) + z_j\left[\mu_z\tau_z + \tau\left(\sum_{\substack{k=1\\k\neq j}}^{n_z}\sum_{i=1}^{N_{jk}}x_{ijk} + \sum_{\substack{k=1\\k\neq j}}^{n_z}\sum_{i=1}^{N_{jk}}x_{ikj} - N_j\mu\right.\right.\right.$$

$$\left.\left.\left. - \sum_{\substack{k=1\\k\neq l}}^{n_z}z_kN_j - \sum_{\substack{k=1\\k\neq j}}^{n_z}I_{jk}\theta_{jk}N_{jk} - \sum_{\substack{k=1\\k\neq l}}^{n_z}I_{jk}\theta_{kj}N_{kj} + 2\sum_{i=1}^{N_{jj}}x_{ijj} - 2N_{jj}\mu - 2I_{jj}\theta_{jj}N_{jj}\right)\right]\right\}.$$

So,

$$z_j|\cdot \sim \; N\left(\frac{\mu_z\tau_z + \tau\left(\overbrace{\sum_{\substack{k=1\\k\neq j}}^{n_z}\sum_{i=1}^{N_{jk}} x_{ijk} + \sum_{\substack{k=1\\k\neq j}}^{n_z}\sum_{i=1}^{N_{jk}} x_{ikj}}^{\substack{\text{sum of the observations with}\\ z_j \text{ on either side, not both}}} - N_j\mu - \sum_{\substack{k=1\\k\neq l}}^{n_z} z_k N_j \right)}{\tau_z + \tau N_j + 4\tau N_{jj}} \right.$$

$$\left. + \frac{\tau\left(\overbrace{-\sum_{\substack{k=1\\k\neq j}}^{n_z} I_{jk}\theta_{jk}N_{jk} - \sum_{\substack{k=1\\k\neq l}}^{n_z} I_{jk}\theta_{kj}N_{kj}}^{\substack{\text{sum of the interactions}\\ \text{with } z_j \text{ on either side, not both}}} + \overbrace{2\sum_{i=1}^{N_{jj}} x_{ijj} - 2N_{jj}\mu - 2I_{jj}\theta_{jj}N_{jj}}^{\substack{\text{terms for the observations with}\\ z_j \text{ on both sides}}} \right)}{\tau_z + \tau N_j + 4\tau N_{jj}}, \frac{1}{\tau_z + \tau N_j + 4\tau N_{jj}} \right).$$

- For τ_z

$$\begin{aligned}
\pi(\tau_z|\cdot) &\propto \tau_z^{a_{\tau_z}-1} e^{-b_{\tau_z}\tau_z} \prod_{j=1}^{n_z}\left(\sqrt{\tau_{z_j}} e^{\frac{-\tau_z(z_j-\mu_z)^2}{2}}\right) \\
&\propto \tau_z^{a_{\tau_z}-1} e^{-b_{\tau_z}\tau_z} \tau_z^{\frac{n_z}{2}} e^{\sum_{j=1}^{n_z}\frac{-\tau_z(z_j-\mu_z)^2}{2}} \\
&\propto \tau_z^{a_{\tau_z}+\frac{n_z}{2}-1} e^{-\tau_z\left(b_{\tau_z}+\frac{1}{2}\sum_{j=1}^{n_z}(z_j-\mu_z)^2\right)}.
\end{aligned}$$

So,

$$(\tau_z|\cdot \sim \; \Gamma\left(a_{\tau_z} + \frac{n_z}{2}, b_{\tau_z} + \frac{1}{2}\sum_{j=1}^{n_z}(z_j-\mu_z)^2\right).$$

- For τ_θ

$$\tau_\theta|\cdot \sim \Gamma\left(a_{\tau_\theta} + \frac{n_\gamma}{2}, b_{\tau_\theta} + \frac{1}{2}\sum_{j=1}^{n_l}\sum_{k=1}^{n_l}(\theta_{jk} - \mu_\theta)^2\right).$$

- For θ_{jk}

There are two cases again; if I_{jk}=0 and if I_{jk}=1.

- If I_{jk}=0,

$$\pi(\theta_{jk}|I_{jk} = 0, \cdot) \sim N\left(\mu_\theta, \tau_\theta^{-1}\right),$$

which is simply the prior.

- If I_{jk}=1,

$$\theta_{jk}|I_{jk} = 1, \cdot \sim N\left(\frac{\mu_\theta\tau_\theta + \tau\left(\sum_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu + z_j + z_k)\right)}{\tau_\theta + \tau N_{jk}}, \frac{1}{\tau_\theta + \tau N_{jk}}\right).$$

- For I_{jk}

$$Pr(I = 0|\cdot) = \frac{1-p}{1 - p + p\exp\left\{\tau\theta_{jk}\left(\sum_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu + z_j + z_k + \frac{1}{2}\theta_{jk})\right)\right\}}.$$

$$Pr(I = 1|\cdot) = \frac{p\exp\left\{\tau\theta_{jk}\left(\sum_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu + z_j + z_k + \frac{1}{2}\theta_{jk})\right)\right\}}{1 - p + p\exp\left\{\tau\theta_{jk}\left(\sum_{i=1}^{N_{jk}} x_{ijk} - N_{jk}(\mu + z_j + z_k + \frac{1}{2}\theta_{jk})\right)\right\}}.$$

3.3.2.3 Asymmetric main and symmetric interaction effects

For the case where the main effects are asymmetric and the interaction effects symmetric, we assume that the effect of the interaction γ_{jk} is treated by the model as the same as γ_{kj}. Thus, at every step of the Gibbs sampler we estimate only the interaction term γ_{jk}.

The model and the full conditionals are the same as in the case where both the main and the interaction effects are asymmetric (3.3.2.1), except for the full conditionals of the interaction effects and the variable selection indicators. The DAG is the same as in Figure 3.2, with one condition added in the indexing of the interaction effects, j should be less or equal to k, $j \leq k$. This is a characteristic of the symmetry assumption.

Using the same notation as in the case of symmetric main effects (3.3.2.2), the updated full conditionals for this case are presented below.

Full conditionals

- For θ_{jk}

  There are two cases again; I_{jk}=0 and I_{jk}=1.

  - If I_{jk}=0,

$$\theta_{jk}|I_{jk} = 0, \cdot \sim \; N\left(\mu_\theta, \tau_\theta^{-1}\right),$$

  which is simply the prior as in the previous cases.

  - If I_{jk}=1

$$w = \begin{cases} \overbrace{\sum_{i=1}^{N_{jk}} x_{ijk} + \sum_{i=1}^{N_{kj}} x_{ikj}}^{\substack{\text{sum of the observations} \\ \text{with } \theta_{jk} \text{ or } \theta_{kj}}} - N_j\mu - N_{jk}(\alpha_j + \beta_k) - N_{kj}(\alpha_k + \beta_j), & \text{if } j \neq k, \\ \overbrace{\sum_{i=1}^{N_{jk}} x_{ijk} + \sum_{i=1}^{N_{kj}} x_{ikj}}^{\substack{\text{sum of the observations} \\ \text{with } \theta_{jk} \text{ or } \theta_{kj}}} - N_{jj}\mu - N_{jj}(\alpha_j + \beta_k), & \text{if } j = k. \end{cases}$$

  So,

$$\theta_{jk}|I_{jk} = 1, \cdot \sim \begin{cases} N\left(\dfrac{\mu_\theta\tau_\theta + \tau w}{\tau_\theta + \tau N_j}, \dfrac{1}{\tau_\theta + \tau N_j}\right), & \text{if } j \neq k, \\ N\left(\dfrac{\mu_\theta\tau_\theta + \tau w}{\tau_\theta + \tau N_{jj}}, \dfrac{1}{\tau_\theta + \tau N_{jj}}\right), & \text{if } j = k. \end{cases}$$

- For I_{jk}

$$Pr(I = 0|\cdot) = \frac{1-p}{1-p+p\exp\left\{\tau\theta_{jk}(w - \frac{1}{2}\theta_{jk})\right\}}.$$

$$Pr(I = 1|\cdot) = \frac{p\exp\left\{\tau\theta_{jk}(w - \frac{1}{2}\theta_{jk})\Big)\right\}}{1-p+p\exp\left\{\tau\theta_{jk}(w - \frac{1}{2}\theta_{jk})\Big)\right\}}.$$

3.3.2.4 Symmetric main and symmetric interaction effects

The last model that we examine using the Kuo and Mallick approach for applying variable selection on two-way Anova models is the case where both the main and the interaction effects are treated as symmetric. We assume that the main effects come from the same distribution and the effect of the interaction γ_{jk} is treated by the model as the same as γ_{kj}.

This case constitutes a combination of the previous two cases, where the full conditionals for τ, z_l, τ_z and μ are the same as in the case where only the main effects are symmetric (3.3.2.2), and the full conditionals for I_{jk} and θ_{jk} are the same as in the case where only the interaction effects are symmetric (3.3.2.3), with the only difference being that w is now defined as:

$$w = \begin{cases} \overbrace{\sum_{i=1}^{N_{jk}} x_{ijk} + \sum_{i=1}^{N_{kj}} x_{ikj}}^{\substack{\text{sum of the observations} \\ \text{with } \theta_{jk} \text{ or } \theta_{kj}}} - N_j\mu - N_j(z_k + z_j), & \text{if } j \neq k, \\ \overbrace{\sum_{i=1}^{N_{jk}} x_{ijk} + \sum_{i=1}^{N_{kj}} x_{ikj}}^{\substack{\text{sum of the observations} \\ \text{with } \theta_{jk} \text{ or } \theta_{kj}}} - N_{jj}\mu - 2N_{jj}z_j, & \text{if } j = k. \end{cases}$$

The DAG in this case is similar to 3.3 with the condition that $j \leq k$ for the interaction effects, as it also happened in the case where only the interactions are symmetric.

An example of the results of these approaches will be presented in Section 7.2, where we examine how MCMC and more specifically the Gibbs sampler works, using synthetic datasets. The models fitted involve two categorical explanatory variables and their interactions on which we apply variable selection using the Kuo and Mallick approach accounting for the levels of asymmetry previously described.

3.3.3 Variable selection using the Horseshoe prior

The Horseshoe prior is the second variable selection technique that we examine. It was analytically presented in Section 2.4.2 and falls into the category of shrinkage priors. It can be used to apply variable selection to models built with algorithms, such as HMC, that are not applicable to problems with discrete variables. Thus, two component discrete mixture priors, like the ones used in the previous section (3.3.2) are not suitable. Other MCMC algorithms, such as the Gibbs sampler, can use spike-and-slab approaches, however it is sometimes suggested that the Horseshoe prior can be more tractable computationally (Piironen and Vehtari, 2017).

Examining the case where both the main and the interaction effects are asymmetric, we want to apply variable selection to the interaction coefficients γ_{jk} using the Horseshoe. Based on the description of the Horseshoe parameterisation (2.11), half-Cauchy prior distributions are used for the hyperparameters determining the variance of γs, therefore:

$$\begin{aligned} \gamma_{jk}|\lambda_{jk}, \tau_{HS} &\sim N(0, \tau_{HS}^2 \lambda_{jk}^2), \\ \lambda_{jk} &\sim C^+(0,1), \\ \tau_{HS} &\sim C^+(0,1). \end{aligned}$$

To avoid confusion with precision τ, we are denoting the hyperparameter of the Horseshoe τ, as τ_{HS}.

The model for the variable selection with the Horseshoe has many similarities to the saturated model presented in Section 3.3.1 and it is defined as:

Model:

$X_{ijk}|\mu, \alpha_j, \beta_k, \gamma_{jk}, \tau$ ~ N($\mu + \alpha_j + \beta_k + \gamma_{jk}$, τ^{-1}), i=1,...,N_{jk},

$N = \sum_{j=1}^{n_\alpha} \sum_{k=1}^{n_\beta} N_{jk}$

μ ~ N(μ_0,τ_0^{-1})

τ ~ $\Gamma(a, b)$

$\alpha_j|\tau_\alpha$ ~ N($\mu_\alpha, \tau_\alpha^{-1}$), j=1,...,$n_\alpha$

$\beta_k|\tau_\beta$ ~ N($\mu_\beta, \tau_\beta^{-1}$), k=1,...,$n_\beta$

$\gamma_{jk}|\lambda_{jk}, \tau_{HS}$ ~ $N(0, \tau_{HS}^2 \lambda_{jk}^2)$

τ_α ~ $\Gamma(a_{\tau_\alpha}$,$b_{\tau_\alpha})$

τ_β ~ $\Gamma(a_{\tau_\beta}$,$b_{\tau_\beta})$

τ_{HS} ~ $C^+(0,1)$

λ_{jk} ~ $C^+(0,1)$

$\tau_\gamma = \frac{1}{\tau_{HS}^2 \lambda_{jk}^2}$

Where α_j is the effect of the j^{th} α, β_k is the effect of the k^{th} β and γ_{jk} is their interaction.

Notation:

n_α is the number of levels of the independent categorical variable α

n_β is the number of levels of variable β

$n_\gamma = n_\alpha \cdot n_\beta$ is the number of levels of interactions

$N_j = \sum\limits_{k=1}^{n_\beta} n_\gamma$ is the number of observations that have α_j

$N_k = \sum\limits_{j=1}^{n_\alpha} n_\gamma$ is the number of observations that have β_k

N_{jk} is the number of observations for which the α is j and the β is k

The corresponding Directed Acyclic Graph (DAG) representing the variables of the model using the Horseshoe is presented in Figure 3.4.

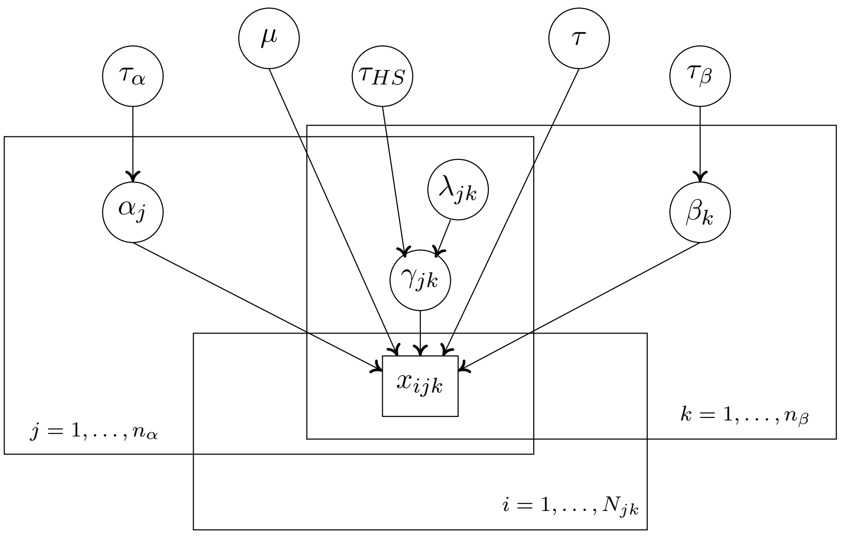


Figure 3.4: DAG for variable selection using the Horseshoe prior

Most of the prior distributions of the unknown parameters are similar to the priors for the saturated model 3.3.1. The difference now is that there is no prior for one common τ_γ, but priors for the newly introduced parameters, τ_{HS} and λ_{jk} instead, and since $\tau_{\gamma_{jk}} = \frac{1}{\tau_{HS}^2 \lambda_{jk}^2}$, the prior for γ_{jk} is adjusted respectively. The likelihood is unchanged, but the joint density is modified to include the new parameters. Finally, the posterior distributions for most parameters are analogous to the ones for the saturated model, thus only the derivation of the posterior distributions for τ_{HS} and λ_{jk} will be displayed.

Priors

- For $\gamma_{jk} \sim \mathrm{N}(\mu_\gamma, {\tau_{\gamma_{jk}}}^{-1})$

$$f(\gamma_{jk}|\mu_\gamma,\tau_{\gamma_{jk}}) \propto \tau_{\gamma_{jk}}^{\frac{1}{2}} e^{-\frac{\tau_{\gamma_{jk}}}{2}(\gamma_{jk}-\mu_\gamma)^2} = \left(\frac{1}{\tau_{HS}^2\lambda_{jk}^2}\right)^{\frac{1}{2}} e^{-\frac{\gamma_{jk}^2}{2\tau_{HS}^2\lambda_{jk}^2}}.$$

So,

$$f(\gamma|\mu_\gamma,\tau_{\gamma_{jk}}) \propto \prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\left(\left(\frac{1}{\tau_{HS}\lambda_{jk}}\right) e^{-\frac{\gamma_{jk}^2}{2\tau_{HS}^2\lambda_{jk}^2}}\right).$$

- $\tau_{HS} \sim C^+(0,1)$

$$f(\tau_{HS}) \propto \frac{2}{\pi(\tau_{HS}^2+1)}.$$

Similarly,

- $\lambda_{jk} \sim C^+(0,1)$

$$f(\lambda_{jk}) \propto \frac{2}{\pi(\lambda_{jk}^2+1)}.$$

So,

$$f(\lambda) \propto \prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\frac{2}{\pi(\lambda_{jk}^2+1)}.$$

Joint density

The joint density is the likelihood multiplied by the priors.

$$\begin{aligned} p(\underline{\underline{x}},\tau,\mu,\underline{\alpha},\underline{\beta},\underline{\underline{\gamma}},\tau_{HS},\underline{\underline{\lambda}}) = {} & \underbrace{(2\pi)^{-\frac{N}{2}}\tau^{\frac{N}{2}} e^{-\frac{\tau}{2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}\sum_{i=1}^{N_{jk}}\left(x_{ijk}-\mu-\alpha_j-\beta_k-\gamma_{jk}\right)^2}}_{\text{Likelihood}} \underbrace{\tau^{a-1}e^{-\beta\tau}}_{\text{prior } \tau} \\ & \underbrace{\tau_0^{\frac{1}{2}} e^{-\frac{\tau_0}{2}(\mu-\mu_0)^2}}_{\text{prior } \mu} \underbrace{\tau_\alpha^{\frac{n_\alpha}{2}} e^{-\frac{\tau_\alpha}{2}\sum_{j=1}^{n_\alpha}(\alpha_j-\mu_\alpha)^2}}_{\text{prior } \alpha\text{s}} \underbrace{\tau_\beta^{\frac{n_\beta}{2}} e^{-\frac{\tau_\beta}{2}\sum_{j=1}^{n_\beta}(\beta_k-\mu_\beta)^2}}_{\text{prior } \beta\text{s}} \\ & \underbrace{\prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\left(\left(\frac{1}{\tau_{HS}\lambda_{jk}}\right) e^{-\frac{\gamma_{jk}^2}{2\tau_{HS}^2\lambda_{jk}^2}}\right)}_{\text{prior } \gamma\text{s}} \underbrace{\tau_\alpha^{a_{\tau_\alpha}-1}e^{-b_{\tau_\alpha}\tau_\alpha}}_{\text{prior } \tau_\alpha} \underbrace{\tau_\beta^{a_{\tau_\beta}-1}e^{-b_{\tau_\beta}\tau_\beta}}_{\text{prior } \tau_\beta} \\ & \underbrace{\frac{2}{\pi(\tau_{HS}^2+1)}}_{\text{prior } \tau_{HS}} \underbrace{\prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\frac{2}{\pi(\lambda_{jk}^2+1)}}_{\text{prior } \lambda}. \end{aligned}$$

Full conditionals

- For τ_{HS}

$$\begin{aligned}\pi(\tau_{HS}|\cdot) &\propto \frac{2}{\pi(\tau_{HS}^2+1)}\prod_{j=1}^{n_\alpha}\prod_{k=1}^{n_\beta}\left(\left(\frac{1}{\tau_{HS}\lambda_{jk}}\right)e^{-\frac{\gamma_{jk}^2}{2\tau_{HS}^2\lambda_{jk}^2}}\right)\\ &\propto \frac{2}{(\tau_{HS}^2+1)\tau_{HS}^{n_\gamma}}e^{-\frac{1}{2\tau_{HS}^2}\sum_{j=1}^{n_\alpha}\sum_{k=1}^{n_\beta}\frac{\gamma_{jk}^2}{\lambda_{jk}^2}}.\end{aligned}$$

- For λ_{jk}

$$\begin{aligned}\pi(\lambda_{jk}|\cdot) &\propto \frac{2}{\pi(\lambda_{jk}^2+1)}\left(\frac{1}{\tau_{HS}\lambda_{jk}}\right)e^{-\frac{\gamma_{jk}^2}{2\tau_{HS}^2\lambda_{jk}^2}}\\ &\propto \frac{2}{(\lambda_{jk}^2+1)\lambda_{jk}}e^{-\frac{1}{\lambda_{jk}^2}\frac{\gamma_{jk}^2}{2\tau_{HS}^2}}.\end{aligned}$$

For both τ_{HS} and λ_{jk}, the resulting posterior density is not of a known form, consequently a Metropolis step will be required for updating the values of these two parameters within the Gibbs sampler.

- For γ_{jk}

$$\gamma_{jk}|\cdot \sim N\left(\frac{\mu_\gamma\tau_\gamma+\tau\left(\sum_{i=1}^{N_{jk}}x_{ijk}-N_{jk}(\mu+\alpha_j+\beta_k)\right)}{\tau_{\gamma_{jk}}+\tau N_{jk}},\frac{1}{\tau_{\gamma_{jk}}+\tau N_{jk}}\right),$$

where $\tau_{\gamma_{jk}}=\frac{1}{\tau_{HS}^2\lambda_{jk}^2}$.

An example of variable selection using the Horseshoe will be presented in Section 7.4, where this method will be examined as an alternative to the Kuo and Mallick approach for applying variable selection to the interactions of a two-way Anova model with random effects.

3.4 Summary

Two-way Anova models are a case of hierarchical models with two independent variables, with each having many levels. If we are also interested in the interactions between the main effects, the corresponding model may have a considerable number of covariates. Using Gibbs sampling to identify the saturated model and incorporate variable selection can help us numerically approximate the parameters of interest. However, deriving the full conditionals can be challenging, depending on the complexity of the problem and the levels of hierarchy in the case of hierarchically structured datasets.

MCMC methods are revolutionary in the field of Bayesian statistics. Nevertheless, due to the increase of volume of data and model complexity, these methods sometimes tend to be computationally expensive. Therefore, in the following chapter we will establish the current state-of-the-art approaches to render these Bayesian inference algorithms more scalable, but at the same time accurate. We will also review the general principles that are followed in order to exploit computing resources efficiently.

Chapter 4

Scalable Bayesian inference

The evolution of Information and Communication Technology (ICT) has led us to the era of "Big Data". Massive amounts of data are generated every day, and their complexity and size create a need for richer models. Modern technologies and scalable solutions aim to provide researchers and corporations with the appropriate tools to analyse data and extract knowledge from them, which can later be used for scientific, business, social and other purposes.

Bayesian methods are widely used for modelling, but their scalability is a challenge. Due to the size of datasets and the complexity of the models, problems like memory constraints, limited storage capacity and excessive computation time arise. Consequently, there is a lot of research to improve these methods, focusing on topics such as the computational resources and programming languages used, the structure of the model, the assumptions made and others.

We will start by mentioning a few reasons that indicate why it is difficult to scale Bayesian approaches in order to cope with the high demands for computational speed and efficiency. We will then analyse some techniques developed over the last years for making MCMC algorithms more computationally efficient and scalable. These techniques mainly fall into two categories: subsampling and divide-and-conquer techniques (Wu and Robert, 2017). After pointing out some aspects of these two approaches and of the ones used specifically in hierarchical modelling we will present some of their limitations.

4.1 Scaling Bayesian methods is a challenge

The concept of scalability has slightly different definitions and interpretations depending on the scientific field. After reviewing the current literature about MCMC techniques, we concluded that there are two notions of scalability, not necessarily uncorrelated. The first is related to how easily MCMC techniques could extend to larger models, either in the sense of model complexity or of sample size. The second is associated with practical speed-up of the sampling process, either by modifying specific parts of the algorithm or by exploiting available technologies. There is an overlapping part in these concepts. Easier extension of the MCMC techniques to bigger models can be achieved if sampling is faster in the algorithmic steps and modern technologies enable processing and storage of large datasets and multidimensional results in an efficient way.

Since the introduction of the Markov Chain Monte Carlo algorithm, the increase of computing capacities have helped Bayesian methods become very popular. However, due to the nature of the algorithms, there are some limitations concerning handling large amounts of data. VanDerwerken and Schmidler (2013) emphasise how MCMC algorithms follow an inherently serial process. Long and time-consuming runs are the result of the iterative character of the algorithms, necessary to obtain convergence to an equilibrium distribution. Moreover, the dependence of the calculations at each step on the estimated values of the previous renders parallelising the process a challenging task. Posterior sampling in MCMC can be computationally intensive and time demanding. The implementation of some MCMC algorithms, such as Metropolis–Hastings, requires a pass over the whole dataset at every iteration while estimating the likelihood. This can be computationally expensive, especially if we deal with a large dataset (Wu and Robert, 2017; Korattikara *et al.*, 2014). Additionally, the complexity of the target density, necessary to calculate the acceptance ratio, may slow down the execution of the algorithms (Banterle *et al.*, 2014). Furthermore, chains appear to have convergence problems when we have high-dimensional posteriors (Rajaratnam and Sparks, 2015). Finally, case-specific problems seem to arise with particular kinds of datasets and models, such as modelling tall data (Bardenet *et al.*, 2014) or hierarchical modelling (Wei and Conlon, 2017).

The above challenges of the algorithms and the calculations involved along with the complex nature of the data structures led to the development of various scaling strategies that try to improve performance by tackling specific parts of the simulation process. Martin *et al.* (2020) study the evolution of Bayesian computation over the years and conclude that all advances in MCMC have a common aim, to explore the high mass region of the target posterior distribution more effectively and consequently improve the estimation accuracy. They also identify three main aims that the scaling methods try to achieve: reduction of the cost per iteration that enables drawing more samples, bias elimination and reduction of dependencies in the chain. The next section aims to present some of the available techniques and outline the latest trends in the field.

4.2 Scaling strategies

According to the current literature there are two main tendencies among the scaling strategies of the MCMC algorithms; subsampling and divide-and-conquer techniques. The first category takes advantage of using random samples of the dataset for the calculations, whereas the second, focuses on parallel processing and spreading the calculations across multiple machines. We devote a section to each technique by gathering and summarising various approaches appearing in the existing literature. We also include a section for some other strategies that either are for more specific problems or do not follow the main two approaches. Next, we focus on methods aiming specifically at more efficient hierarchical modelling. Finally, we refer to some limitations of the scaling techniques.

4.2.1 Subsampling strategies

Having to sweep through the whole dataset at every iteration of an MCMC algorithm can be computationally expensive. Consequently, there are several strategies developed based on using a subset of the initial dataset at every iteration, a process which is called subsampling (Angelino *et al.*, 2016; Bardenet *et al.*, 2014). Below we cite some proposed implementations based on subsampling. The general idea is to decrease the complexity $\mathcal{O}(N)$ of regular MCMC, where N is the sample size, by operating on subsets of the original dataset (Quiroz *et al.*, 2018). This approach aims to improve the performance of the algorithm by optimising the algorithm itself (Korattikara *et al.*, 2014).

Angelino *et al.* (2016) prove why subsampling works. For the problems that we mostly deal with, the data are conditionally independent given the parameters θ. When this is the case, not only can the posterior be expressed in terms of the product of the likelihood and the prior, but also the likelihood can be decomposed into a product of the probability that we observe each data-point given θ as follows:

$$\pi(\theta|\mathbf{x}) \propto \pi(\theta)\pi(\mathbf{x}|\theta) = \pi(\theta)\prod_{n=1}^{N}\pi(\mathbf{x}_n|\theta). \tag{4.1}$$

According to Angelino *et al.* (2016), when N is large the above factorisation (4.1) can be used in MCMC algorithms where updates will only depend on subsets of the data and not the entire dataset. It is also pointed out that these subsets can be used to form an unbiased MC estimate of the log likelihood and the log joint density. This derives from the fact that the log likelihood is the following sum:

$$\log\,\pi(\mathbf{x}|\theta) = \sum_{n=1}^{N}\log\,\pi(x_n|\theta), \tag{4.2}$$

that can be approximated with the following sum, using a random subset of m terms:

$$\log\,\pi(\mathbf{x}|\theta) \approx \frac{N}{m}\sum_{n=1}^{m}\log\,\pi(x_n^*|\theta), \tag{4.3}$$

where $\{x_n^*\}_{n=1}^m$ is a uniformly random subset of $\{x_n\}_{n=1}^N$ and $m < N$. This approximation leads to an unbiased estimate of the log joint density which is the product of the log likelihood and the log prior density.

MCMC algorithms are based on sampling, and contain some kind of accept/reject step to evaluate the sampled value and determine their next move. Hence, some of the methods developed focus on applying subsampling strategies on the accept/reject step. Another group of techniques introduces auxiliary variables and some alternative methods add noise to stochastic optimization techniques to incorporate subsampling (Angelino *et al.*, 2016). Finally there are some other approaches that do not strictly fall in these categories.

• Reformulation of the accept/reject step

Metropolis–Hastings is one of the most widely used MCMC algorithms and there is an effort to reformulate the way its accept/reject step is estimated by incorporating the idea of subsampling. According to Korattikara *et al.* (2014), this step, also referred to as MH test, can be very time-consuming. This is due to the fact that it necessitates calculation of the likelihood over thousands of data points. Using all data points at every iteration can be inefficient. MCMC has a finite amount of computational time T to calculate its output. During this time there are two features that affect the stepsize of the proposal (ϵ) that is involved in the calculation process; the bias and the variance. The bias is mostly related to the burn-in period, when the algorithm is sampling from the wrong distribution, and has a decreasing tendency. The sampling variance is related to the random nature of the process. They point out that using subsamples at the accept/reject step can lead to unbiased likelihood estimators that will, however, have high variance and vice versa. To eliminate this bias-variance trade-off, they develop an approximate MH test that introduces some bias in the stationary distribution, but compensates it by reducing the variance more quickly and decreasing the computational complexity. Based on the same principle, Bardenet *et al.* (2014) also develop a method that accounts for the bias-variance trade-off. Angelino *et al.* (2016) use the term "Adaptive subsampling" when referring to this category of methods. Most of these methods apply uniform subsampling at each iteration. Another approach, suggested by Maire *et al.* (2018), involves a subsampling mechanism based on summary statistics. Their algorithm, Informed Sub-Sampling MCMC (ISS-MCMC), extends the state space $\mathit{\Theta}$ with an n-dimensional vector (U_k), that at every iteration (k) indicates the subset of the original N data points used, hence $U_k \subset \{1, ..., N\}$ where $n << N$. The sub-samples are assigned a distribution that reflects their similarity to the full dataset and summary statistics are used as a similarity measure.

• Usage of auxiliary variables

Firefly Monte Carlo (FlyMC) is an approach proposed by Maclaurin and Adams (2014). In this strategy a binary auxiliary variable for each observed data-point is used to indicate at each iteration which of the data-points are used for estimating the posterior distribution.

• Noise addition to stochastic optimisation

Welling and Teh (2011) developed a subsampling technique called Stochastic Gradient Langevin Dynamics (SGLD) algorithm. It is based on another MCMC algorithm, the Metropolis-adjusted Langevin algorithm (MALA), which uses *stochastic optimisation*. SGLD computes gradients on various subsets and uses them to approximate the true gradient over the whole dataset. It introduces noise to the stochastic gradient estimator in order to optimise the un-normalised density. Then, by decaying this noise they manage to accept all proposals and therefore never evaluate the acceptance probability (Angelino *et al.*, 2016).

• Other approaches

Based on the Pseudo-marginal MCMC, where a MH algorithm uses auxiliary random variables (not introduced to apply subsampling but already included) to form the likelihood estimates in order to make the intractable distribution tractable, comes the approach proposed by Quiroz *et al.* (2018). At each MCMC iteration they use a small random subset of the data and two types of control variates to obtain an unbiased estimate of the exact log-likelihood.

Linear mixed models, composed by fixed and random effects, are widely used in many real-world applications. Tan *et al.* (2018) investigate ways to model high-dimensional Linear Mixed Models (LLM) using a kernel matrix, computed using the subsampled Hadamard transformation. They also introduce approximate variance components to estimate the fixed-effects coefficients. Their approach results in computational complexity that depends sublinearly on the number of covariates p.

Despite their differences, all of the approaches mentioned above are based on the idea that at each iteration, a random subset of the dataset is used in order to speed-up the algorithms and make them more efficient. Some of the sub-sampling techniques mentioned, allow the sub-sample size to vary (Korattikara *et al.*, 2014; Bardenet *et al.*, 2014), whereas others use a fixed sub-sample size (Maire *et al.*, 2018). For the first category of methods there is an associated computational cost stochastically defined at each iteration, due to the calculation of the data mini-batch size used for this particular iteration, that could potentially outweigh the efficiency gain of performing calculations on a subset of the original dataset. Since there is not a gold standard scaling strategy applicable to all statistical problems, it is important that more than one methods are explored to identify the best approach for each situation, if efficiency is of interest.

The next category of scaling methods aims to achieve speed-up by taking advantage of parallel computing resources.

4.2.2 Divide-and-conquer strategies

With the evolution of technology and computing science, multi-core and cloud technologies that allow for parallel processing became extensively used for various algorithms. It is a challenge to manage to embed them in the MCMC class of algorithms in order to improve mixing and distribute the computational load more efficiently. A lot of research has been done in this area, and the methods mentioned in this section aim to take advantage of modern computing resources to make MCMC algorithms more scalable. Methods based on the idea of spreading the calculations across multiple machines are called divide-and-conquer methods (Wu and Robert, 2017). According to Wilkinson (2005) there are two main strategies for parallelising MCMC; one is based on using multiple processors to run multiple MCMC chains in parallel and the other is based on parallelising a single MCMC chain. Some more recent techniques do not necessarily fall into this categorisation since they aim to parallelise specific parts of the algorithms.

The introduction of fast graphics processing units (GPU) managed to decrease the processor bottleneck, but memory and disk bottlenecks were still an issue. Splitting the data across machines could be a solution. However, the communication cost among multiple machines can be quite expensive, so it must be minimised. Scott *et al.* (2016) introduced Consensus Monte Carlo (CMC) that aims to tackle the communication cost problem. The algorithm divides the data across many nodes, where each node samples independently from the posterior distribution given the data that are allocated to it. So, a separate Monte Carlo algorithm runs on each machine and only at the end the posterior draws are combined to a 'consensus chain', which is the weighted average of the separate chains, that reflects the results from all the worker nodes of the distributed system. Several variations of the consensus Monte Carlo approach have been developed. The limitation of CMC is that it is exact only with Gaussian posterior and sub-posterior distributions (Scott, 2017*a*). Neiswanger *et al.* (2014) propose a similar approach, sampling from an estimate of the subposterior density product, which is proportional to the full-data posterior, using kernel smoothing. Unlike consensus Monte Carlo, their algorithm is asymptotically exact for any posterior, but has high computational complexity (Bumbaca *et al.*, 2017). Another approach that again connects the independent draws at the end is followed by the Weierstrass Sampler which is in fact two alternative algorithms: refinement sampling and rejection sampling, both based on application of a Weierstrass transformation on the sub-posteriors (Wang and Dunson, 2014). Instead of sampling from independent sub-posteriors as in the previous methodologies, Wu and Robert (2017) suggest sampling from rescaled sub-posteriors, across many machines. They recentre each sub-posterior to their common mean and then average to take the approximation of the true posterior.

VanDerwerken and Schmidler (2013) introduce an asynchronous parallelisation approach where running an arbitrary number of independent and parallelisable copies of a Markov chain and then combining the results can accelerate the process of estimating the equilibrium. This can be achieved by partitioning the parameter space Θ into j partitions whose sampling average of the target distribution combined with a weight w will result in an unbiased estimate of the unknown parameter. Another technique that does not involve exchange of information between the nodes over which the workload is spread until the end of the algorithm is called "Powered Embarrassing Parallel MCMC", where the data posterior density is again the product of the posterior densities of different subsets of the data that run on different machines (Li *et al.*, 2017). On the other hand, the "Generalized Elliptical Slice Sampling" (Nishihara *et al.*, 2014) functions by allowing information sharing between the chains. This approach uses latent Gaussian models and takes advantage of Gaussian priors to enable faster mixing. The target distribution is reframed accordingly. The transition operator carries information about the shape of the target distribution and parallelism is used to choose the tuning parameters of the scale mixture of Gaussians.

Another group of methods tries to combine optimisation techniques with MCMC. It is based on the alternating direction method of multipliers (ADMM) algorithm which aims to solve optimisation problems by breaking them into smaller pieces. The variables of interest, which are subject to constraints, are split into a pair of variables. This enables simplification of the objective function, which is now a sum of the data fitting term and a regularisation function (Vono *et al.*, 2018). In the case of the MCMC, it is the target posterior distribution that is split and the subposteriors are distributed across nodes (Rendell *et al.*, 2020).

A technique that aims to parallelise a single chain is implemented by Banterle *et al.* (2014). It divides the acceptance step into several parts by partitioning the "prior $\times$ likelihood" term into a product. The less costly components are evaluated first and compared to a sample from a Uniform distribution. If one of the components is rejected the algorithm does not proceed in calculating the remaining components. Moreover, their algorithm takes advantage of a computer science technique which is called *prefetching*. It can calculate, in parallel, operations that are expected to be needed soon. In this case, the potential posterior values are computed in advance based on a binary tree representation of the probable outcomes. The left branches represent the acceptance and the right the rejection. A prerequisite is that the random-walk steps and the sequence of uniform samples driving the accept/reject steps will be simulated beforehand in order to build the tree.

According to Wilkinson (2005), parallelising MCMC can get complex due to the dependencies between the various parts of the model. Therefore, another approach that aims to parallelise a single MCMC chain is based on the identification of groups of variables that are conditionally independent. A natural way to identify which tasks can be carried out independently of one another is to use a Directed Acyclic Graph (DAG), that can give a natural representation of the terms of the model and show the conditional independences. Then, these variables can be updated in parallel on the available processors. Their summary statistics can also be computed in parallel. Variables that depend on other variables are updated sequentially to ensure correctness. This strategy can extend to complex models as well, as long as the conditional independence assumption holds.

When it comes to MCMC algorithms that take advantage of parallelisation there is a range of possible solutions with two extreme ends. The one involves running a large number of independent Markov Chains in parallel but according to Nishihara *et al.* (2014) this approach will enable running longer chains in the same time and improve the accuracy but not the mixing or the speed of convergence. At the other end, there is one chain that parallelises the transitions and potential expensive operations that can be calculated in parallel. Additionally, there is a variation concerning the level and amount of communication among the nodes, with many approaches trying to minimise this communication during the process and allowing it only for the final calculations.

4.2.3 Other strategies

Some strategies aim specifically to models that have intractable likelihood functions. Friel *et al.* (2016), introduce control variables to approximate the likelihood, and at the same time reduce the estimator variance. They use K forward-simulations, with K being equal to the number of the available cores, which can run in parallel and reduce the computational time.

A lot of research has been done for more specific kinds of models. Johndrow *et al.* (2018) study ways to use block updating to accelerate the mixing of the global precision parameter and decrease the computational cost per step of MCMC when the number of predictors is greater than the number of observations $p > N$, and especially when the vector of coefficients β, with $\beta \in \mathbb{R}^p$, is sparse. The idea is to use the Horseshoe prior and sample from the joint distribution of blocks of the unknown parameters in a specific order. Papaspiliopoulos *et al.* (2018) explore scalability in another kind of model, the crossed random effects models. They use a block Gibbs sampler that updates in a block the levels of a given factor, and specifically each factor level separately, conditionally on the rest, as well as a collapsed version of this algorithm that samples again in blocks the levels of each factor after having integrated out the global mean first.

An approach based on "blocking" and the use of the topology of DAGs as well as the sparsity of the conditional independence while working with models containing a large number of latent (unobserved) variables is proposed by Wilkinson and Yeung (2004). They show how using sparse matrix algorithms and the canonical parameterisation of a Multivariate Normal Distribution can lead to more convenient schemes that can be used to implement more efficient MCMC algorithms. Another study on latent Gaussian models uses nested Laplace approximations. A three-step algorithm is proposed to approximate posterior marginals in this type of model. The first step involves using Laplace approximation to approximate the posterior marginal of θ. The second step computes the Laplace approximation of $\pi(x_i|y, \theta)$, for selected values of θ, and the third step combines the previous two. This algorithm results to computational gains (Rue *et al.*, 2009).

4.2.4 Strategies for hierarchical modelling

Hierarchical modelling is used when the data are organised in groups and we want to identify the within and between group variabilities. There are many problems of that form and, due to the nature of the data, the methodologies mentioned earlier can be applied but are not adequate for hierarchical modelling. Their limitation is that they have a significant cost either in accuracy of results or in the computational gain (Bumbaca *et al.*, 2017). Consequently, there is a need for further improvement since, specifically for hierarchical models, it is necessary to take into consideration the multilevel structure of the datasets.

One approach that can be used for scalable hierarchical modelling is an extension of the consensus Monte Carlo idea mentioned earlier (4.2.2). This version of the algorithm is implemented in two stages. For each stage, the full dataset (Y) is partitioned into S shards, with each shard being on a different machine, such that all of the observations for a group are in the same shard. The first stage simulates both the common parameter among all groups, denoted as θ, and all the group-level parameter draws. At every iteration of the MCMC, it only combines the S collections of θ and drops the rest. The common parameter draws from that stage are combined as a product of S subposteriors ($p(\theta|Y) \propto \prod_s(\theta|Y_s)$) and used at the second stage. The second stage draws the group-specific parameters in an embarrassingly parallel (i.e. without communication among the machines) manner using the product of subposteriors from the previous stage (Scott, 2017*b*).

Wei and Conlon (2017), suggest partitioning the data grouped by factors, since most parameters are group-specific, and then use parallel computing and split the groups on different cores. They also propose a two-stage approach that uses Metropolis-Hastings. In this case, after partitioning the full dataset by groups, the first stage involves the group-specific parameters' independent parallel estimation. The second uses the posteriors produced from the first stage as proposal. In a similar approach Bumbaca *et al.* (2017) construct an estimator for the posterior predictive distribution of the group-specific parameters in the first stage, and use the estimator as the prior distribution in the second stage, with both stages being embarrassingly parallel. Finally, another algorithm for parallelising Bayesian hierarchical models is implemented in MultiBUGS, which is a version of BUGS[1] aiming to speed up posterior inference of this specific type of model. MultiBUGS uses a directed acyclic graph, where each node (n) is a component DAG G=(V_G, E_G), where V_G includes all the stochastic parameters of the model, including the hyperparameters, and the observations, and E_G represents the arrows. Due to the assumption of conditional independence the full joint distribution of all quantities denoted as V, is a factorisation of each node v, given its parents $p(V) = \prod\limits_{v \in V} p(v|pa(v))$. MultiBUGS identifies a partition of a set of stochastic parameters (S(V)) for which each partition component contains only mutually conditionally independent parameters which enables them to be sampled in parallel. So, this algorithm parallelises the evaluation of the likelihoods formed as products and the sampling of conditionally independent sets of parameters (Goudie *et al.*, 2018).

4.2.5 Limitations of scaling approaches

All the approaches mentioned earlier aim to make MCMC algorithms more efficient and less prone to delays due to increasing complexity of the models. However, there are some limitations that have to be considered when applying one of these strategies.

According to Bumbaca *et al.* (2017) some of the algorithms work efficiently only when they deal with Gaussians and mixture component distributions can increase the computational complexity. This is also supported by Wei and Conlon (2017). Additionally, some methods use kernel density estimation that becomes infeasible when dealing with higher dimensions. Furthermore, sometimes proper priors for the full dataset might be improper for the subsets that can lead to improper posteriors for these subsets. Angelino *et al.* (2016) also point out that in most cases more parallelism means less accuracy. Moreover, although there is an overall speed up not many algorithms try to speed-up the burn-in process (Neiswanger *et al.*, 2014). Finally, from a time consumption perspective communication between the machines can be inefficient, since multiple machine coordination can be computationally expensive (Scott *et al.*, 2016).

4.3 Summary

With the rapidly growing scales of statistical problems and datasets, there is high demand to change the implementations of existing MCMC algorithms in a way that they can be

[1]Software for Bayesian analysis (BUGS, 1989).

more scalable, efficient and at the same time accurate. There are two main approaches to this problem. One is subsampling, where random subsets of the original dataset are used for the calculations. The second approach involves distributed computation, described often as divide-and-conquer techniques. These strategies use parallel processing and computing resources. Some implementations of these strategies demand communication between the mini batches of data stored on different machines, a factor that can increase the communication cost. Consequently, there have been efforts to minimise this cost by creating "embarrassingly" parallel algorithms that demand no communication until the end when the chains are combined to mimic draws from the full data posterior distribution. Most approaches have limitations. Some parameters to consider before deciding which scaling strategy is more suitable for a problem are the structure of the dataset, the model to be used, the final aim of the research and all the trade-offs and the properties that are important for the specific study.

MCMC is a powerful simulation technique and although there are computing tools and libraries developed for running its algorithms, in some cases where the model is non-standard it is necessary to practically reimplement the algorithm. Some of the programming languages that are widely used for probabilistic programming are R, Python, Java, Matlab and C. Most of those languages are imperative or object-oriented with embeded features of other programming paradigms as well. Over the last few years however, functional programming has been gaining ground. The functional programming paradigm is based on binding the structure of the program in a mathematical functions style and it has started being used in probabilistic programming as well. Examples of functional programming languages are Haskell and Scala. The latter is the language of choice for this project. In the following chapter we will describe what functional programming is, its main principles, its advantages and disadvantages, and how it can be used in statistical computing.

Chapter 5

Functional programming

“Computational statistics” or “statistical computing” are two terms, often used interchangeably, referring to applying computing science to statistics in order to deal with demanding problems. In the era of “Big Data”, where large scale and complex datasets can be difficult to handle, computer resources are used to increase statistical power and enable faster and more accurate statistical inference and prediction, data analysis and graphical representation, even for analytically intractable problems. Some of the latest technological trends, such as machine learning and artificial intelligence, are based on statistical methods and there is a lot of research in the field of improving the algorithms used and making them faster and more scalable, as we examined in Chapter 4. Therefore, it is important to select from among the various programming languages and computing environments, the tools and technological resources that correspond better to the needs of each application.

Programming languages, based on their features, can be classified in categories which are called programming paradigms. Among others, one common programming paradigm is the imperative, in which the code is a series of instructions that modify the variables in the memory and describe how the program operates. Another widely used paradigm is object-oriented programming (OOP). Its core attribute is the “objects” that contain various fields of data defining the object’s state and methods that can modify these states. Most programming languages, such as Java, C++ and Python, support multiple paradigms since they borrow features from more than one paradigms.

Functional programming constitutes another programming paradigm that is continually gaining ground. Unlike OOP, it is based on the idea that the structure of a program is formed with functions that avoid changing any state. Instead, when data are being processed, a new representation of them is being created and returned. A pure functional programming language is Haskell, and an impure language is Scala, which falls into the category of multi-paradigm, with object-oriented and functional programming features.

The extent to which a Scala program will borrow features from each programming direction depends on the programmer. Scala was introduced in 2001 by Martin Odersky at the École Polytechnique Fédérale de Lausanne (EPFL). It runs on the Java Virtual Machine and is becoming more popular in Big Data analytics due to its flexibility, parallelisation properties and conciseness.

In the following sections we will present the principles of functional programming and provide examples in Scala in order to better explore this programming style and showcase how it can be used for statistical computing and scaling of widely used statistical methodologies.

5.1 Principles of functional programming

At the core of functional programming is the idea that a program is composed of many smaller functions, each following the principles of this paradigm that facilitate writing properly structured code. In the following sections we will present a definition of functional programming and we will introduce some of its main concepts along with some important language-specific features of Scala that facilitate writing programs using this programming approach. A simple tool to evaluate expressions in Scala is an interactive command line interpreter shell called the Scala REPL. Furthermore, integrated development environments (IDE), such as IntelliJ IDEA provide Scala plugins. Both tools were used to produce the code snippets that follow. Additionally, some programmers prefer using text editors such as Emacs.

5.1.1 Pure functions, immutable values and referential transparency

The functional programming paradigm is based on the idea of mathematical functions. In the literature there are a lot of definitions for it but as Alexander (2014) emphasises, all of them include the following statements:

A functional program should:

1. consist of *pure functions*
2. only include immutable values

"Pure functions" constitute one of the most important concepts of functional programming. The term is used to denote functions that do not have any *side effects*. For a function to be considered pure, it has to not interact with the environment outside its local scope. Some examples of interaction could be reading user input from a file or database, writing output to a memory unit or a web service, or mutating a global variable. This property ensures that a pure function only depends on its input parameters and its internal code.

Hence, for a given input parameter x the returned output y will always be the same for an infinite number of function calls. An example of a pure function is the `max` method of a `List`. For a given `List` as an input, it will always return the same maximum value. On the other hand, an example of a function that is not pure is the method `nextInt` of a `Random` class that will return a new random value every time it is called.

The second statement of the definition, *immutable values*, is another important feature of functional programming. It is based on the algebraic notion incorporated in the programming perspective, that a variable can be substituted with its value at any point in the program as it retains the same value, since nothing has allowed its alteration throughout the execution of the program. Immutable variables are denoted as `val` in Scala and there is always a compile error if the programmer tries to reassign a value to an immutable variable throughout the program. The only variables that can be reassigned are declared with the keyword `var`.

Unlike single variables, of which the mutability is determined by just using the appropriate keyword, using `val` for a collection does not guarantee that the collection cannot change. There are specifically immutable collections for this purpose. This is because using `val` with a mutable collection ensures that the whole collection cannot be reassigned, however individual elements can be changed and new elements can be added and deleted since the collection is mutable. Complete immutability is achieved only if a collection is immutable and the keyword `val` is used. Only in that case the collection cannot change in any way.

Functional programmers liken pure functions to algebraic equations. They combine these functions to create series of expressions that have a clear input and output in order to solve a programming problem the same way mathematical functions are combined to solve a mathematical problem. Function composition is a key part of functional programming as the creation of a chain of functions leads to structured and clear pieces of code that follow the logical order of problem solving.

Pure functions and immutability offer *Referential Transparency (RT)*. Referential transparency is a term used in functional programming to describe the property that an expression, which is called referential transparent, can be replaced with its resulting value without changing the program semantics. For example, there are three immutable variables `y, a` and `b`, with the association `val y = a + b`. Whenever the referential transparent expression `a + b` is used, it can be replaced by `y` and vice-versa, and this action does not affect the output of the program (Odersky, 2008). This "substitution principle" does not hold for imperative programs.

5.1.2 Higher-order functions

A feature that is not part of the core definition of functional programming, but constitutes an important element of Scala, is *Higher-Order functions (HOFs)*. In this part firstly we will give a simple example to introduce the idea of higher order functions and then we will present some HOFs that are long established and widely used in functional programming.

5.1.2.1 HOFs – a simple example

In Scala, functions are treated as first class citizens and they can be passed as arguments to other functions or be returned from other functions, such functions are called Higher-Order functions (Odersky, 2016). To make the notion of HOFs more clear, we will use a simple example to demonstrate how a function can be a variable too, in order to be passed to other functions or be returned from other functions based on the definition of HOFs. Consider a function that checks if an integer is even. In Scala it could be written as:

```
val isEven = (i:Int) => i % 2 == 0
```

This function checks the modulo of an integer and returns a Boolean value true if the modulo is 0 indicating that the integer is even. The function reference has:

- the name "`isEven`"
- the input parameter "`i`" of type "`Int`"
- the function body "`i % 2 == 0`" that is the algorithm that defines its functionality
- the type "`Int => Boolean`" declares a function that transforms the input parameter of type `Int` into a "`Boolean`"

If we want to call this function in Scala to check if number 4 is even we write: `isEven(4)` and we would get the result `res: Boolean = true`.

We can expand the example above to show how the function `isEven` can be passed as an argument to a HOF. Consider a list of integers (`List[Int]`):

```
val listofInts = List(1,2,3,4,5)
```

In Scala there is a function named "`filter`" that is supported by, among others, the immutable collection `List`, and selects all the elements that satisfy a specific condition passed as an argument in it. Since the function `isEven` acts as a variable and can be passed in another function as an argument, we can write:

```
listofInts.filter(isEven)
```

that will select only the even elements of the list `listOfInts` and return:

```
res: List[Int] = List(2,4)
```

So, “`isEven`” is a reference to a function and can be passed as an argument in the HOF `filter` (Alexander, 2014). This simple example captures the idea, but does not illustrate the power of higher order functions. In bigger problems HOFs provide reusable functionality which can decrease problem complexity. Some other HOFs widely used in functional programming are `map` and `flatMap`, that will be analysed in the following section.

5.1.2.2 map, flatMap, foldLeft

Applying a function to all elements of a list is a common operation. For all the core data structures in Scala there is a Higher Order Function, named `map` that as `filter` accepts a function as an argument (function input parameter), but unlike `filter` that returned a subset of the initial list, `map` applies the function to all the elements of the original list. It returns a new list that has the same size as the initial, containing the resulting values after having applied the function input parameter.

To make the use of `map` more clear, consider a list of integers (`List[Ints]`):

```
val listofInts = List(1,2,3,4,5)
```

Suppose that we want to transform every integer of the `listofInts` into its double. Using the function map on the list:

```
val x = List(1,2,3,4,5).map(_*2),
```

results in:

```
x: List[Int] = List(2,4,6,8,10).
```

A `List`, or any other data collection, that has been subject to transformation by `map` will not always be of the same data type as the initial collection. For example, it might be necessary to transform a `String` to an `Int` corresponding to its length, so if we have multiple strings stored in a `List`, a `List[String]` will be transformed to a `List[Int]`. Hence, the type signature of the `map` function of the Scala `List[A]` class is of the form:

```
map[B](f: A => B): List[B]
```

The part “`f: A => B`” indicates that `map` takes as argument a function that transforms an element of type `A`, denoting the type of the initial `List`, to an element of type `B`. Then “`: List[B]`” denotes that the transformed elements are stored in a `List` of type `B`. The ability to abstract over the type of the data structure is due to an important feature supported by many languages, including Scala, which is called polymorphism. A more detailed explanation of polymorphism will be provided in Section 5.1.3.

Another important Higher Order Function is `flatMap`. Its type signature for a `List` is defined as:

```
flatMap[B](f: A => List[B]): List[B]
```

This is translated as `flatMap` taking as argument a function that transforms a variable of type `A` to a `List[B]` and returns a `List[B]`.

An example that will display both the functionality of `flatMap` and the difference between `map` and `flatMap` includes a `List[Int]` and a method that takes an integer and returns a `List` with the initial and an increased integer:

```
val listReduced = List(1,2,3)
def increment(x:Int) = List(x, x+1)
```

Applying `map`:

```
listReduced.map(x => increment(x))
```

results in: `List(List(1,2), List(2,3), List(3,4))`

However, applying `flatMap`:

```
listReduced.flatMap(x =>increment(x))
```

results in: `List(1,2,2,3,3,4))` which is the flattened result of `map`.

This difference between `map` and `flatMap` does not seem very important for this simple example but is extremely useful for complex programs and applications.

The final HOF presented in this section is `foldLeft` that is commonly used for traversing a collection from left to right and applying a binary operation to its elements in order to collapse them to one resulting value. The type signature of `foldLeft` is:

```
def foldLeft[B](z: B)(op:(B, A)=>B): B
```

`foldLeft` accepts two arguments, `(z: B)` which is a seed of type `B`, and a binary operator `op:(B, A)` that returns a value of type `B`. The result of `foldLeft` is again a value of type `B`. For example, in the case of `x: List[Int] = List(1,2,3,4,5)`
`val sum = x.foldLeft(0)(_+_)` results in `sum =15`.
`foldLeft` starts with the value 0 and adds it to the the first element of the list which is 1. The resulting value is then added to the second element and the process continues until the list is exhausted. Then the final result is returned.

So far we have presented how some of the most common Higher Order Functions are used on collections where they are already embedded. Programmers can define their own `map` when they do not use one of the language's standard collections. In the following part we present how `map` is defined for the immutable Scala collection `List`, examples of which we used in this section, however without providing more details about its implementation. Similarly, the implementation of `map` can extend to other custom collections based on their functionality and characteristics.

5.1.2.3 Implementation of `map` for `List`

Before presenting how `map` can be defined for `List`, in order to modify every element of the list while maintaining its structure, it is worth providing some more details about this core Scala collection.

The immutable linked lists represent a sequence of data where each element contains a link to its successor. A `List` is an *algebraic data type* which means that it is a data type that is defined by one or more data constructors, each of which may contain zero or more arguments. A `List` consists of two elements, the `head`, which is its first, and the `tail` that contains all the other elements, or it can be an empty list `Nil`. Consequently, based on the definition of `List` in Scala's standard library after being subject to alteration for the case of `Int`, there are two data constructors as shown below (Chiusano and Bjarnason, 2017):

```scala
sealed trait List[+Int]
case object Nil extends List[Nothing]
case class Cons[+Int](head: Int, tail: List[Int]) extends List[Int]
```

A Scala `trait` is a way to define object types such as the `List` that we refer to. According to the definition of `List` a new `List` of four integers (1,2,3,4) can be created as:
`val list = Cons(1, Cons(2, Cons(3, Cons(4, Nil))))`

A common operation on a list is to apply a function `f` to every element while maintaining its structure. As mentioned earlier this is feasible by using a `map` function, that for the `List` could be defined by:

```scala
def map[A,B](l: List[A])(f: A => B): List[B] = l match {
   case Nil => Nil
   case Cons(h, t) => Cons(f(h), map(t)(f))
}
```

The function `map` takes a list of elements and a function operating on them. If the list is empty it returns an empty list (`Nil`). If the list has elements, the function `f` is applied to the `head (h)` which is followed by a call of `map` on the rest of the elements `tail(t)`, all stored in a new list. This procedure uses *recursion* since every step involves a call of `map` to itself and ends when the final element of the list is processed.

Higher order functions are highly significant in functional programming as they allow for abstraction over the control flow. They can be combined and act as binders of functions in order to support the proper structure of a functional program and implement the main idea of the functional programming paradigm, that is, as Alexander (2014) points out "to make the code look and work like algebraic expressions". These functions can be extremely powerful when combined with functions intended to solve problems of real world applications.

5.1.3 Polymorphism

Parametric polymorphism is an important concept in programming that allows programmers to write more generic code in order to avoid repetition of code that provides the same functionality to different types. It refers to the ability of a function to be applied to arguments of many types that do not need to be predefined in the declaration of the function (Odersky, 2016). This concept leads to the redefinition of `List`, presented in the previous part to embed the idea of using `A` as the placeholder for any type that a list could consist of. The more abstract definition of `List` is (Chiusano and Bjarnason, 2017):

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

With this definition, as long as all elements have the same type, the contents of the `List` could be of any type, e.g. `Double`, `Int`, `String`. Otherwise, the collection would lack type safety and we would not be able to recover information about the type of any of the elements of the `List`, because it would be `List[Any]`. Note that when declaring the type parameters in `List` the `+A` is used. This signals that `A` is a positive parameter (covariant) of `List` denoting that subtypes of `A` can be used as elements of the `List`, such as `Nothing` which is a subtype of all types and therefore `Nil` can also be considered as a `List`. Another type of polymorphism is called ad-hoc polymorphism. It is also referred to as overloading, and allows for the creation of multiple functions with the same name but different implementations.

Concepts such as pure functions, immutable values, Higher Order Functions and polymorphism, that were presented in this section, are fundamental building blocks of functional programming. More complex concepts are often based on these principles and are the tools to build solutions to real world applications. In the following section we will introduce *Streams*, a Scala collection with particular characteristics.

5.2 Streams

Collections in Scala can be classified into two big categories; the strict and lazy collections. The former category refers to the collections that evaluate their elements when they are created, while the latter involves the collections that will only evaluate their elements when they are needed.

Lists constitute a data structure that is widely used in Scala. They are not lazy evaluated, which means that when a list is created or transformed, all of its elements are computed. However, in cases where there is complex computation, a lazy collection such as `Stream[A]`, whose elements are computed only when we iterate through the structure, can prove very beneficial. Unlike lists, streams can be created and transformed without computing their elements and this can allow for the creation of a potentially infinite collection (ScalaDocs, 2020).

An example where streams can be advantageous is when using a Markov Chain Monte Carlo algorithm, examined analytically in Section 2.2. Streams, as lazy data structures, can capture the logic of the convoluted steps and numerous iterations. Actual evaluation of the stream and extraction of its values is achieved through its conversion to a regular collection, after truncating it to one of finite length. To demonstrate how a `Stream[A]` can be used in MCMC and specifically in the implementation of a Gibbs sampler, we will use a simple linear regression problem.

Consider the model: $y = \alpha 1 + X\beta + \epsilon$ with n observations, p covariates and X an $n \times p$ matrix, and suppose that all p covariates will be used for predicting y. The unknown parameters that we want to estimate are α, β and τ. Programmatically we can organise the updated values of all the parameters at the end of each iteration/update, using a particular type of class called *case class* named `FullState` and defined as:

```scala
case class FullState(alphaTau: Vector[Double], bcoefs: Vector[Double])
```

where `alphaTau` is a vector consisting of two values, the updated α and τ and `bcoefs` represents the covariates.

Suppose that there is a function `calculateNextState` that, as illustrated below, updates the unknown variables by updating α and τ and then the β coefficients through two functions named `nextAlphaTau` and `nextCoefs` respectively. These functions use the full conditional distributions of the variables of which the implementational details are not of interest at the moment.

```scala
def calculateNextState(fstate: FullState): FullState = {
    val latestAT = nextAlphaTau(fstate)
    val latestFullyUpdatedState = nextCoefs(latestAT)
    latestFullyUpdatedState
  }
```

This function takes as argument an object of the case class `FullState`. It will be the initial value of the variables to initialise the Gibbs sampler during the first iteration and the latest updated state at every other iteration of the algorithm. The returned value of the function is the fully updated state of the unknown variables, again of type `FullState`.

An approach that does not make use of streams would require a loop to iterate through the algorithm for as many times as we would like to run the MCMC. At the end of every iteration the new state with the latest updated variables should be appended to a list that stores the results. Alternatively, we could use recursion for a more functional implementation. However, for a stream-based approach we specify a function `streamStates`, that enables the production of the infinite stream, using the native Scala function `Stream.iterate` with the type signature:

```
def iterate[A](start: A)(f: A => A): Stream[A]
```

`iterate` accepts the start value of the stream and the function `f` that is repeatedly applied, and returns the infinite sequence of values: start, f(start), f(f(start)),... .

```
def streamStates(fState: FullState): Stream[FullState] =
Stream.iterate((fstate))(fstate => calculateNextState(fstate))
```

Finally, we have to declare the number of the elements of the stream that we would like to keep, extract the values by converting the stream to another regular collection such as a list or a vector, and get a sequence representing the results of each update of the MCMC as follows:

```
// Initialise the parameters
val initAT = Vector(0.0, 1.0)
val initBetas = Vector.fill(p)(0)
val updatedState = FullState(initAT, initBetas)
val iterationsNo = 100000

val results = streamStates(updatedState)
       .drop(100) // do not evaluate first 100 iterations, burn-in
       .take(iterationsNo) // returns the iterationsNo first elements of this
           Stream
       .toVector // converts the Stream to a Vector
```

Thinning can also be implemented using `filter`. In smaller examples this implementation works perfectly and efficiently, however when the number of variables or iterations increases, as with using other implementations not involving streams, memory issues might appear. This problem can be easily solved by saving the results to a file while sampling.

In the above example we used the embedded Scala Stream collection. Alternatively, there are libraries, such as Akka Streams, FS2 and Monix, that define their own stream data types with various properties and functions in order to facilitate large scale processing from various sources. Stream-based approaches, and especially when using MCMC, can lead to clean solutions that can handle issues such as burn-in and thinning in an elegant and easy to understand way.

5.3 Functional programming and category theory

An important consideration in programming is code re-usability that aims to reduce redundancy and processing time and efficiently make use of the available resources. Apart from various rules that apply when trying to write reusable code there are some embedded features in programming languages that help programmers towards this goal. Collections such as *Lists*, *Tuples* and *Options* share common characteristics and abstractions, combinations of which can lead to polymorphic, cleaner and more structured reusable code. Some of these abstractions originate from a field of Mathematics known as *Category Theory.*

In this section we present how category theory has influenced functional programming. After introducing some of the principal terms of this field we will show how they are embedded into the functional programming paradigm. Finally, we will showcase how *Monads* are used in functional programming to achieve the main aim of this programming style: to compose functions for a more concise and structured programming solution.

5.3.1 Basic terminology in category theory

Understanding the structures and applications in programming that are influenced by category theory does not require having in-depth knowledge of the corresponding mathematical definitions. However, comprehension of some principal ideas of this mathematical discipline can prove beneficial. Therefore, in this part we introduce definitions for some fundamental concepts in category theory.

5.3.1.1 Category

A *category* (C) is an algebraic structure that consists of a collection of *objects* and a collection of arrows showing the relations between them, called *morphisms.* The set of morphisms from object x to y in a category C is denoted as: $hom_C(x, y)$, and the set of objects as obj(C). Figure 5.1 illustrates an example of a category with three objects.

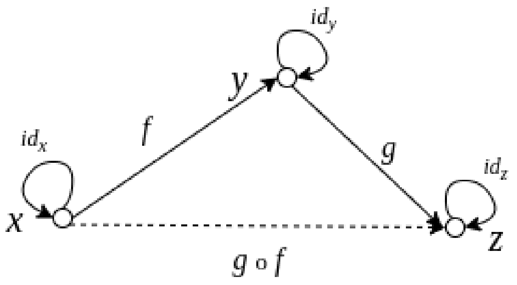


Figure 5.1: A category of three objects

A morphism is expressed as: $x \xrightarrow{\mathrm{f}} y$, which can also be written as: $f : x \to y$. Adjacent morphisms can be composed, for example: $x \xrightarrow{f} y \xrightarrow{g} z$, can be written as: $x \xrightarrow{g \circ f} z$, meaning that their composition defines a relation between x and z. This composition is associative so $h \circ (g \circ f) = (h \circ g) \circ f$ (nLab, 2020). Furthermore, for every object of the category there is an identity arrow that connects the object with itself, hence there is a loop in every object in Figure 5.1.

Definition. A category C is a collection of objects and the morphisms between them, such that for the objects x, y and z, and the morphisms $f : x \to y$ and $g : y \to z$ (Law, 2019; Grozev, 2016):

1. There must exist a morphism $h : x \to z$ which is the composition of f and g, $h = g \circ f$
2. Composition of morphisms must be associative, so $h \circ (g \circ f) = (h \circ g) \circ f$
3. Each object in C must have an identity morphism $id_x : x \to x$
4. For every morphism $f : x \to y$, $id_y \circ f = f = f \circ id_x$. Thus, identities must be neutral to composition.

5.3.1.2 Functor

Transformations between two categories A and B are called *functors* (F). A functor maps every object and every morphism of the first category to the second, whilst preserving the structure. It is denoted as: $F : A \to B$. An example of a functor is presented in Figure 5.2.

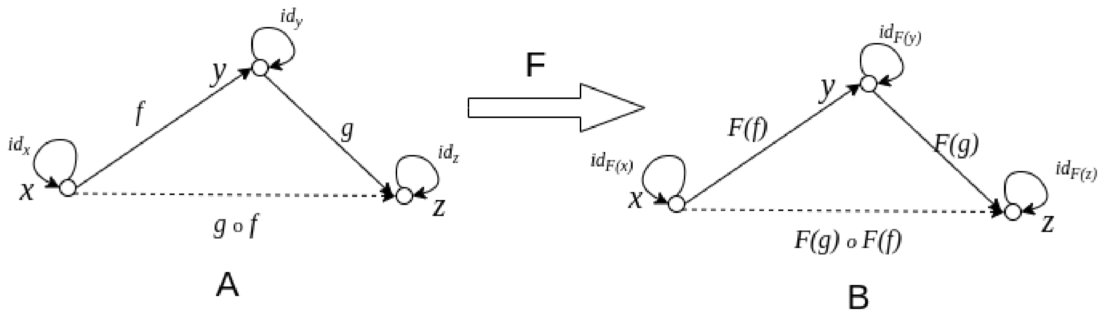


Figure 5.2: F: A functor from category A to B

In this graph we can see that (Grozev, 2016):

- Each identity arrow in A is preserved in the transformed category B
- Every association between the objects of the category A is preserved in B

- $F(g \circ f) = F(g) \circ F(f)$

Definition. If A and B are two categories, then a functor $F : A \to B$ maps an object x in A, to an object $F(x)$ in B and each morphism $f : x \to y$ in A to a morphism $F(f) : F(x) \to F(y)$ such that:

1. $F(id_x) = id_{F(x)}$ for each x in A
2. $F(g \circ f) = F(g) \circ F(f)$ for all morphisms f and g in A

A special case where a functor F transforms a category to itself is called an endofunctor and is denoted as $F : A \to A$. Set is the category of sets and set functions. Functional programming is mainly concerned with the category of data types and functions which is very similar to the category of "Set". Therefore, in this context all functors are considered endofunctors of type $F : \text{Set} \to \text{Set}$ (Law, 2019).

5.3.1.3 Natural Transformation

For every category there are morphisms between the objects that show their relationship and how one object can be transformed to another. Transformations exist between categories as well, through the functors that were presented at the previous part. Respectively there are also morphisms between functors, which are called *natural transformations* and preserve the structure of the categories.

Suppose that there are two categories A and B, two objects x and y from category A and two functors $F, G : A \to B$, then the natural transformation η ($\eta : F \Rightarrow G$) is a family of morphisms between F and G. It represents an assignment to every object x in A of a morphism $\eta_x : F(x) \implies G(x)$ in B, such that for any morphism $f : x \to y$ in A the combinations of relationships, illustrated in Figure 5.3, commute in B (nLab, 2020).

$$\begin{tikzcd}
F(x) \arrow[r, "F(f)"] \arrow[d, "\eta_x"'] & F(y) \arrow[d, "\eta_y"] \\
G(x) \arrow[r, "G(f)"'] & G(y)
\end{tikzcd}$$

Figure 5.3: Natural transformation $\eta : F \Rightarrow G$ along with its components

Definition. If F and G are two functors between two categories A and B ($F, G : A \to B$) then a natural transformation $\eta_x : F(x) \implies G(x)$ is a family of morphisms such that (Law, 2019):

1. For every object $x \in A$, the component of η at x, $\eta_x : F(X) \to G(X)$, is a morphism in B
2. For every morphism of category A $f : x \to y$, $\eta_y \circ F(f) = G(f) \circ \eta_x$

The second statement of the definition is visualised in the commutative diagram in Figure 5.3, that shows how the two sequences of morphisms, with $F(G)$ as the starting point, lead to the same result $G(Y)$.

5.3.1.4 Monad

Another term that is widely used in functional programming is *Monad* that derives again from category theory.

Definition. A monad (T) on a category C is an endofunctor $T : C \to C$ with two natural transformations:

- a unit ($\eta : Id_C \to T$), which is a natural transformation of the identity functor to the monad
- $\mu : T \circ T \to T$, which is a natural transformation from C to C

and the monadic laws (Law, 2019):

1. $\mu \circ T\mu = \mu \circ \mu T$
2. $\mu \circ T\eta = \mu \circ \eta T = Id_T$

$$\begin{tikzcd}
TTT \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & TT \arrow[d, "\mu"] \\
TT \arrow[r, "\mu"'] & T
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[dr] & TT \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl] \\
& T &
\end{tikzcd}$$

(b)

Figure 5.4: Commutative diagram expressing (a) the first condition (b) the second condition of the monad laws (nLab, 2020).

In the following part we will demonstrate how these the basic concepts of category theory are embedded and implemented in functional programming.

5.3.2 Application of category theory in functional programming

In this part we will demonstrate how functional programming uses category theory principles to implement structures that are at the core of this programming paradigm. These structures, with monadic characteristics, are used to write composable blocks of code. The examples included are implemented in Scala. The theoretical background is based on the library Cats (2019*b*) for Scala, designed to provide abstractions based on category theory.

5.3.2.1 Type constructor and higher-kinded types

In Section 5.1.2 we saw that one way to calculate the sum of all the elements within a `List` is using `foldLeft`. Similarly we can calculate the sum of other structures as well, such as `Trees` or `Vectors`. To capture this commonality in a concept where the type of a structure is not important as long as it supports a `foldLeft` method, we use a trait:

```
trait Foldable[F[_]]{
def foldLeft[A,B](strc: F[A])(z:B)(f: (A,B) => B): B
//foldRight combines the items from right to left
def foldRight[A,B](strc: F[A])(z:B)(f: (A,B) => B): B
          ⋮
}
```

The trait includes all the methods that will have to be implemented in the class/classes that extend it, according to their functionality.

The argument of `Foldable` is `F[_]` and it is called *type constructor*. Type constructors take a type parameter as indicated by the underscore. `Foldable` that takes a type constructor as an argument, is referred to as a *higher-order type* or *higher-kinded type* with respect to the higher-order functions mentioned in Section 5.1.2.

Type constructors and higher-kinded types are also a feature that enables polymorphism and will be used often in the following parts.

5.3.2.2 Type classes

Type classes are a powerful tool in functional programming and they enable ad-hoc polymorphism. Features from Category theory are embedded in some functional programming languages, such as in Scala, by the use of type classes. One of the simplest type classes in Scala is `Monoid`. More complex type classes are `Functors`, `Applicatives` and `Monads`, that gradually build their functionalities by extending other type classes.

Type class: Monoid

Suppose that we were interested in collapsing a list in various cases:

```
def sumIntegers(list: List[Int]): Int = list.foldRight(0)(_ + _)

def sumDoubles(list: List[Double]): Int = list.foldRight(0.0)(_ + _)

def concatenateStrings(list: List[String]): String = list.foldRight("")(_ ++ _)

def unionSets[A](list: List[Set[A]]): Set[A] = list.foldRight(Set.empty[A])(_
    union _)
```

All the above examples follow a similar pattern. They start with an initial value that depends on the type (0 for `Int`, empty string " " for `String` etc.) and they combine the elements using a function (+ for `Int`, ++ for `String` etc.). It is easy to avoid using unnecessary re-implementation of the same function for each case with abstraction. This can be implemented in Scala as follows:

```
trait Monoid[A]{
   def empty: A
   def combine(x: Int, y:A): A
}

//Implementation for Int
implicit val intAddMonoid: Monoid[Int] = new Monoid[Int]{
   def empty: Int = 0
   def combine(x: Int, y:Int): Int = x + y
}

//Implementation for String
implicit val stringConcatMonoid: Monoid[String] = new Monoid[String]{
   def empty: Int = ""
   def combine(x: String, y: String): String = x ++ y
}

// A common function for all types
def collapse[A](list: List[A])(implicit A: Monoid[A]): A =
    list.foldRight(A.empty)(A.combine)
```

This implementation is an example of a *type class* that involves:

- a trait defining the signatures of the functions that each case will have to implement based on the properties and functionality of its type
- creation of an instance of the trait for each type and implementation of its functions
- implementation of functions that are common for all types.

For monoids, there are two function definitions, an associative binary function and an identity. Having declared the instances of the trait and the monoid parameter in `collapse` as implicit, the compiler will know which `empty` and `combine` implementation to call based on the type parameter of the `List` argument given in `collapse`. A compile error will occur if `collapse` is called with a `List` with a type parameter for which there is no monoid implementation.

Type class: Functor

In Section 5.3.1 we saw that a *functor* is a transformation between two categories that maps every object and every morphism from the first category to the second, keeping the structure intact. In Scala a `Functor` is in accordance with this definition and is a type class that abstracts over the type constructors that can be mapped over. Type constructors are the types that take one type argument `F[_]`, such as `List` and `Option`. In Scala, `Option[T]` is a structure that acts as a container for zero or one element of a given type. It can be either `Some[T]` or `None`. As in the case of the `Monoid` type class, for the `Functor` first we define a trait and then we provide with implementations for various type constructors.

```
trait Functor[F[_]]{
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

//Implementation for List
implicit val functorList: Functor[List] = new Functor[List] {
  def map[A, b](fa: List[A])(f: A => B): List[B] = fa map f
}

//Implementation for Option
implicit val functorOption: Functor[Option] = new Functor[Option] {
  def map[A, b](fa: Option[A])(f: A => B): Option[B] = fa match {
    case None => None
    case Some(a) => Some(f(a))
  }
}
```

To illustrate how the `List` Functor can transform various types we use the example in Figure 5.5. Every type `A` becomes `List[A]` e.g. `Int => List[Int]`. The `List` Functor also affects the morphisms/functions by allowing the lifting of every function `A => B` into the function `F[A] => F[B]`, for example `toDouble => map(toDouble)`, so that `List[A] => List[B]`. This can be coded by the use of a `lift` function added to the definition of the `Functor` trait:

```
def lift[A,B](f: A => B): F[A] => F[B] =
  fa => map(fa)(f)
```

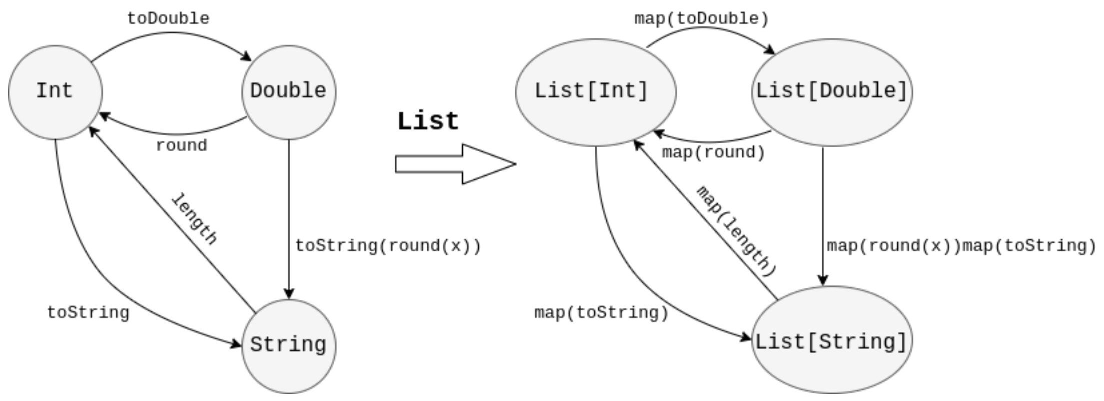


Figure 5.5: An example for the `List` Functor

Type class: Applicative

The `Applicative` type class extends `Functor` and adds a `pure` function that “lifts”/“wraps” a value into the type constructor. Hence, for `List` the `pure` function creates a singleton list, and for `Option`, it creates `Some(_)`. The trait `Applicative` is defined as:

```
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
}
```

`Applicative` also includes an `ap` function that allows application of a function wrapped in the context `F[_]`, to a value wrapped in the context `F[_]`, by unwrapping both the function and the value, applying the function to the value and then wrapping the resulting new value in the same context. Supposing that we have an `Option[Char => Double]` and an `Option[Char]` to which we want to apply the function, a typical `map` would not be able to achieve that, therefore `ap` can be very powerful in this case.

`Applicative` also supports composition. If `F` and `G` have `Applicative` instances, so does their composition `F[G[_]]`. Furthermore it offers `mapN` (`N` = 2,...,22) functions that can apply an n-argument function to n wrapped in the context values and preserve the outer context. However this requires knowing in advance the exact number of values that will have to be used. There are cases where this is unknown, such as when reading from a database or when there is user generated input. This problem is solved by another type class, `Traverse`, that is actually a `Functor` and includes a function `traverse` to walk through a structure.

Type class: Traverse

`Traverse` is a type class that provides an abstraction over the structures that can be traversed over, such as `List`s and `Tree`s. The trait is defined as follows:

```
trait Traverse[F[_]] {
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
}

// Example implementation for List
implicit val traverseForList: Traverse[List] = new Traverse[List] {
  def traverse[G[_]: Applicative, A, B](fa: List[A])(f: A => G[B]): G[List[B]] =
    fa.foldRight(Applicative[G].pure(List.empty[B])) {(a, acc) =>
      Applicative[G].map2(f(a), acc)(_ :: _)
    }
}
```

For the example of the `List` as `F`, `traverse` will walk through the list `fa`, using the function provided `f`, that returns an `Applicative G[B]`, suppose an `Option[B]`, and will return a `List[Option[B]]`, instead of `Option[List[B]]` that would be returned if `map` was used.

Type class: Monad

A `Monad` is a type class in functional programming that is widely used. It enables function composition and computation sequencing through its supported function implementations. Furthermore, since one of the main principles in functional programming is avoiding side effects such as input/output actions, some monads, such as the `IO Monad` are not only wrapped around these actions in order to clearly define that a side effect is taking place, but also they perform the "unsafe" action at the end of the program.

The type class `Monad` extends `Applicative` with the addition of a new function `flatten` which takes a value in a nested context (e.g. `F[F[A]]`) and after joining the context together it returns the value wrapped in a single context (e.g. `F[A]`). In Section 5.3.1 where we defined some of the main concepts of category theory for monads we mentioned two characteristics. The first was unit ($\eta : 1_A \rightarrow T$), which is provided in Scala by `Applicative` through `pure`, and the second was $\mu : T \circ T \rightarrow T$, supported in Scala by `flatten` also referred to as `join`

```
def flatten[A](fa: F[F[A]]): F[A]
```

However, it is established in functional programming that we express `Monad` in terms of `pure` and `flatMap`. So, instead of `Applicative`, `Monad` extends `Functor` which provides `map` and implements its own `pure` and the newly introduced `flatMap` as shown below:

```
trait Monad[F[_]] extends Functor[F] {
  def pure[A](x: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

where `flatMap` could be defined using `map` and `flatten` as follows:

```
def flatMap(fa:F[A])(f: A => F[B]) =
  flatten(map(fa)(f))
```

Both `map` and `flatMap` are very powerful in functional programming as they facilitate the process of composing functions as part of solving bigger problems. To demonstrate function composition consider three functions, `div`, `log` and `sqr` defined as:

```
def div(x: Int, y: Int): Option[Int] =
  if (y == 0) None else Some(x/y)

def log(x: Int): Option[Double] =
  if (x <= 0) None else Some(math.log(x))

def sqr(x: Double): Option[Double] =
  Some(x*x)
}
```

To ensure correctness of the constraints imposed on the first two functions we use an `Option` that returns `None` for `div` if the denominator equals 0, and for `log` if the input is negative or 0. The third function does not have any constraints but has to return an `Option` as well, so as to be composable along with the other two functions, and to match the output of the one with the input of the other. The three functions can be combined as:

```
def functionComposition(x: Int, y: Int ): Option[Double] =
  div(x,y).flatMap(z => log(z)).flatMap(i => sqr(i))
}
```

In `functionComposition`, `flatMap` is used, and not `map`, in order to avoid cumulative creation of `Option`s every time that a new function is applied. This leads to a clean result which is an `Option[Double]`. In Scala there is also another way of expressing sequence comprehensions, which is called *for-comprehension* and for our example `functionComposition` would be redefined as:

```
def functionCompositionForComp(x: Int, y: Int ): Option[Double] = for {
  a1 <- div(x,y)
  a2 <- log(a1)
  a3 <- sqr(a2)
  } yield a3
}
```

For-comprehensions as shown in `functionCompositionForComp` have the form: `for - enumerators - yield e`. An enumerator can have various forms. It can be of the general form "pattern ←expression" which introduces new variables and is then called a *generator*. The right side of the generator can be a container of elements resulting in the left side of the arrow after iterating over all the elements of the container. The right side can also be a monad, as in the above example, where `a1` ←`div(x,y)`, and `div(x,y)` is an Option monad. In this case, the arrow is extracting the value from the monadic context and assigns it to the variable on the left. An enumerator can also be a *definition* of the form "pattern = expression". Then, the left side consists of a variable that is bound to the value of the right side of the equality. Finally, an enumerator can be a filter of the form "`if(expression)`". The final part of a for-comprehension `yield e` is what is produced by this block of code (Alexander, 2014).

A for-comprehension is considered to be *syntactic sugar* in Scala for a series of `flatMap` calls followed by a `map`. It is easier for humans to understand the clearer representation of a for-comprehension rather than a sequence of `flatMaps` and a `map`, that the compiler translates the for-comprehension to. Even though in the above example `map` is not visible in `functionComposition` the compiler adds it at the end as:
`div(x,y).flatMap(z => log(z)).flatMap(i => sqr(i).map(c => c))`

To clarify the sequence of `flatMaps` followed by a final `map` suppose that we want to add the `Double` 2.0 following the other operations, then `functionComposition` and `functionCompositionForComp` would be modified as follows:

```
def functionComposition(x: Int, y: Int): Option[Double] =
  div(x,y).flatMap(z => log(z)).flatMap(i => sqr(i).map(z => z + 2.0))
}
//OR
def functionCompositionForComp(x: Int, y: Int): Option[Double] = for {
  a1 <- div(x,y)
  a2 <- log(a1)
  a3 <- sqr(a2)
  } yield a3 + 2.0
}
```

In the example above, function composition was possible because all functions returned the same monadic instances. Combining functions in this way would not have been feasible if they were returning different monads, for example, if `div` was returning an `Option[Int]` and `log` a singleton list `List[Double]`. However, there are other ways that enable mixing of monadic contexts inside for-comprehensions such as *monad transformers*.

5.3.2.3 Kleisli composition and monad transformers

One of the cornerstones in functional programming is the ability to break complex problems into smaller pieces and use function composition to combine the solutions to those subproblems. In the cases where the returned types and the arguments of the functions composed are classified among the main data types supported by the functional programming language there are usually trivial ways to create the pipeline that will produce the integrated solution. For instance, in Scala `andThen` and `compose` are used to join functions together. Suppose that there are two functions:

```
def f[A, B]: A => B
def g[B, C]: B => C
```

Their composition can be achieved by defining a function `composeAll`

```
def composeAll[A, C]: A => C = f andThen g
//equivalent to g compose f, f.andThen(g) and g.compose(f)
```

However, if the functions were monadic, and returned the values in a context, they should be chained with a for-comprehension or with a sequence of `flatMaps` and a `map` as shown in the previous part.

```
def f[A, B]: A => Option[B]
def g[B, C]: B => Option[C]

def composeAllForComp[A, C]: A => Option[C] = (a: A) =>
  for {
    b <- f(a)
    c <- g(b)
  } yield c

def composeAllFlatMap[A, C]: A => Option[C] = (a: A) =>
  f(a).flatMap (b => g(b))
```

Note that again the final `map` is not necessarily explicitly written because the final value is not subjected to any further computations. However, the compiler does translate the body of `composeAllFlatMap` as being:

```
(a: A) => f(a).flatMap(b => g(b)).map(c => c)
```

While the above ways to chain monadic functions are extensively used in functional programming, there is also another way to compose functions of this kind by using a wrapper around the function `A => F[B]`, known as `Kleisli[F[_], A, B]`. Some functional programming languages support this data type natively, but for Scala it is supported by libraries like Cats and Scalaz. If we can call `flatMap` on `F[A]`, hence `F[_]` has a `flatMap[F]` instance `Kleisli` takes the following form as defined in Cats (Cats, 2019*a*):

```
import cats.FlatMap
import cats.implicits._
case class Kleisli[F[_], A, B](run: A => F[B]) {
   def compose[Z](k: Kleisli[F, Z, A])(implicit F: FlatMap[F]): Kleisli[F, Z, B]
       = Kleisli[F, Z, B](z => k.run(z).flatMap(run))
```

`Kleisli` contains a function `compose` that enables function composition in the same way as for functions which return values not in the monadic context. In the following example:

```
def charToInt(c: Char): Option[Int] =
  if (c >= 'a' && c <= 'z') Some(c.toInt) else None

def log(n: Int): Option[Double] =
  if (n <= 0) None else Some(Math.log(n))

def charToIntThenLog(c: Char): Option[Double] =
  for {
    a <- charToInt(c)
    b <- log(a)
  } yield b

//Call the function by passing the argument of the first function to be called,
    for example:
charToIntThenLog('a')
```

The two functions `charToInt` and `log` return an `Option` and the function `charToIntThenLog` combines them. In that example the result of parsing a `Char` to `Int` within the range ['a','z'] will always be a positive number. Using `Kleisli` leads to the following implementation:

```
def charToInt: Kleisli[Option,Char,Int] =
  Kleisli((c: Char) => if (c >= 'a' && c <= 'z') Some(c.toInt) else None)

def log: Kleisli[Option,Int,Double] =
  Kleisli((n: Int) => if (n <= 0) None else Some(Math.log(n)))

def charToIntThenLog: Kleisli[Option,Char,Double] =
  log.compose(charToInt)

charToIntThenLog('a')
```

In all the examples illustrated so far the returned types were values in a monadic context of the same type, e.g. `Option` or `List`. There are ways to compose the `Monad` instance of a data type with any other `Monad` instance using *monad transformers*. A monad transformer for `Option` is `OptionT[F[_], A]` and allows us to combine the properties of `Option` with any other `F[_]`, such as a `List`. It is a wrapper around `F[Option[A]]` but more convenient to use. Consequently, another way to think of `Kleisli` is as a *monad transformer* for functions, as it allows us to combine the effects of a function with the effects of any `F[_]`.

5.3.3 Category theory and functional programming summary

Structures based on category theory are embedded in functional programming in the form of type classes to create cleaner and reusable code. Monads are a commonly used example of a type class. It not only enables function composition through the implementation of `map` and `flatMap`, but also ensures that potential side effects will be handled in a safe way. In the following section we will demonstrate how probabilistic programming languages are based on type classes to implement Bayesian inference while hiding implementational details from the users.

5.4 Functional programming and probabilistic programming

The Bayesian paradigm is a long-established reliable way to build models that define uncertainty about problems, after taking into account the judgment of statisticians and field experts, reflected in the prior distributions, and the available data. However, building a Bayesian model can be challenging and complicated, and therefore a prohibiting factor for exploring new models that need a deeper knowledge of Bayesian statistics. This barrier could be reduced by a programming paradigm, along with the equivalent programming languages, that would aim to specify and define probabilistic models at a higher-level, with the use of traditional programming ideas, in order to automate the process of making inference for these models. This notion defines the concept of probabilistic programming; a programming approach that does not rely on detailed comprehension neither of the programming aspects nor of the Bayesian inference algorithms (Law and Wilkinson, 2019).

Probabilistic programming languages (PPLs) allow the declaration and description of inference problems whilst retaining a level of abstraction. The aim of probabilistic programming is to enable users to fully specify a probabilistic model, by writing code that defines the data, the prior distributions and the likelihood, and set some algorithmic parameters, without obligatory further involvement in the sampling process. Then, the program automatically uses this model specification for inference. PPLs internally evaluate and computationally solve the denoted problems, resulting in probability distributions for the unknown parameters using the provided data. Observations are used to condition the distribution of the unknown variables, reducing our uncertainty about them. In order to achieve automation, to the highest degree, of the steps of statistical modelling, PPLs combine the fields of machine learning, statistics and computer science. There are numerous domain specific languages (DSLs) used for probabilistic programming such as Figaro (Pfeffer, 2015), STAN (Stan, 2015), JAGS (Plummer, 2017), BUGS (Lunn *et al.*, 2009), Rainier (Bryant, 2018), and many others. Some are based on an already existing language and some others have their own syntax. Typically a PPL includes widely used features of common programming languages, such as primitive operations and conditional statements.

Probabilistic programming enables users to write programs that facilitate Bayesian inference and answer different queries residing in three main categories: *expectation, sampling* and *support*. The expectation of a function f is the mean of f over the distribution, whereas sampling refers to drawing a value from a distribution. If a large number of samples are drawn repeatedly we can approximate both the expectation and the shape of the distribution. Finally, support identifies with non zero probability the subset out of the entire sample space, from which a sample came (Ramsey and Pfeffer, 2002).

Usually, probabilistic programs can be translated to graphical models represented with graphs, where the nodes correspond to variables and operations. Variables act as inputs to operations/functions of which the output becomes the input of other operations/functions. This translation is important because some existing inference algorithms are expressed with regard to graphical models, and therefore these algorithms can be more easily written in a probabilistic programming language (van de Meent *et al.*, 2018).

Similarly to deterministic programming languages, probabilistic languages can be developed within the imperative, functional, and logical frameworks. Here, we will focus on the functional programming paradigm. Functional probabilistic programming languages and libraries incorporate the fundamental functional programming concepts developed in the previous sections. A foundational property in functional programming is referential transparency. As described in Section 5.1.1, this property conveys that a function f, when applied to a value x, will always return the same value $y = f(x)$. This notion, however, contradicts the purpose of probabilistic events, that due to their non deterministic nature, cannot guarantee the same outcome every time they happen. A solution to that problem could be the encapsulation of probabilistic values in a data type known as *probability monad* (Erwig and Kollmansberger, 2006).

In the following parts we will provide a high-level presentation of the probability monad and its usage in writing probabilistic programs in a functional scheme. We will then introduce Rainier, a probabilistic programming library for Bayesian inference in Scala, to show how the modelling API is decoupled from the inference algorithm.

5.4.1 Probability monad

In Section 5.3.2 we presented type classes, such as Monads. They are particularly useful in functional programming as they have some properties that enable easier implementation of functionalities. Moreover, they facilitate the process of composing smaller functions, are easier to reason about, implement and test, in order to a form a more structured and correct larger program. The idea of a probability monad was first established by Lawvere in 1962 (Lawvere, 1962), and was further extended by Giry (1982) 20 years later. The extension, application and usage of monads in functional probabilistic programming, via a probability monad, is proved to be an important component. Functional programming languages and libraries that support monads have the ability to use the monadic properties offered by FP languages and broaden the range of capabilities a user has in order to define and manipulate a model.
Recalling from 5.3.2 a `Monad` in Scala is defined as:

```scala
trait Monad[F[_]] {
  def pure[A](x: A): F[A]
  def map[A, B](fa: F[A])(f: A => B): F[B]
  def flatten[A](fa: F[F[A]]): F[A]
}
```

where `flatMap` is expressed in terms of a `map` followed by `flatten` or `join`

```scala
def flatMap(fa: F[A])(f: A => F[B]) =
flatten(map(fa)(f))
```

Embedding the monadic notion in the probabilistic programming framework is achieved through some main building blocks. Firstly, there is a data type `Prob[T]`, representing probabilities and probability densities. Secondly, there is a data type `Distribution[T]` representing probability distribution over objects of type `T`, along with some primitives consisting of widely used distributions such as uniform and normal. Thirdly, there is a monadic instance for `Distribution` that enables construction and composition of distributions. Finally, it is necessary to have an implementation that would allow the probabilistic system to observe the data and condition the distributions to update the prior and result in the posterior distributions, and a way to sample from a `Distribution` (Scibior *et al.*, 2015).

These concepts can only be applied if probability distributions form a monad, hence they need to support the functions in the definition of the `trait Monad`:

- `pure` - Represents the unit η_X: `X -> Distribution(X)`. It takes a value `x` ($x \in X$), where X is a measurable space, and encapsulates it in a distribution. It actually represents the Dirac distribution, which results in a constant distribution with only one possible outcome with probability 1.

- `map` - maps a function `f: T =>U` over a probability distribution `Distribution[T]` and this results in an object of type `Distribution[U]`. It is a transformation that preserves semantics.
- `flatten` - Represents the multiplication:
  μ_X: `Distribution(Distribution(X))-> Distribution(X)`. It can be depicted as a tree with two levels where `join` can have a dual representation depending on the nature of the problem. In applications regarding simulation or simulation based optimisation that use domain specific algorithms, it can be represented as sampling from a distribution of distributions, which can be achieved using a pseudo-random number generator. In applications related to linear and dynamic programming that require a more holistic approach and modelling of the entire distribution, this distribution of distributions can be flattened after considering all the paths and multiplying all the probabilities (Jelvis, 2017).

In probabilistic programming there are several ways to represent a probability distribution, in this part we focus on two. The first representation is with a function mapping objects to their probabilities, as shown in the implementation of `Dist[T]` (Listing 5.1). The second option is as a sequence of samples that leads to the sampling based distribution `Rand[T]`. A large number of samples will approximate `Dist[T]`.

```scala
//General distribution
trait Dist[T] {
   def probOf(t: T): Double
}
//Sample based distribution
trait Rand[A]{ self =>
   def draw(): A
   def sample(n: Int): List[A] = List.fill(n)(draw)
   def map[B](f: A => B): Rand[B] = new Rand[B] {
      override def draw = f(self.draw)
   }
   def flatMap[B](f: A => Rand[B]): Rand[B] = new Rand[B] {
      override def draw = f(self.draw).draw
   }
   def pure[A](a: A):Rand[A] = new Rand[A] {
      override def draw = a
   }
}
```

Listing 5.1: Traits to represent distributions

Calling the function map on a distribution applies a function to all the elements of that distribution. For example, the support [0,1] of a uniform distribution can be altered by multiplying each element with a number a: uniform.map(x => x * a).

The use of flatMap can be very helpful in Bayesian statistics to add variables to an existing model and to combine the prior distribution of each parameter with the likelihood in order to get the posterior distribution, after having observed the data. Using map to fit the model leads to two levels of distributions Rand[Rand[T]], one for the prior distribution and one for the likelihood. However, using flatMap removes the extra layer of distribution by first applying the function and then flattening the result. In order to take advantage of the monadic properties, Rand must also have an implementation for pure, map and flatMap. Then, the Rand trait can have a draw function as abstract in order to be implemented by the new object representing a distribution according to its characteristics.

These traits (Listing 5.1) can be used to define specific distributions. For example, the discrete probability distribution for a Bernoulli trial with two possible outcomes "Success" and "Failure", can be written using a case class:

```
case class Bernoulli(p: Double) extends Dist[Boolean] {
  def probOf(t: Boolean): Double =
    if (t) p else (1 - p)
}
```

Listing 5.2: The Bernoulli distribution

According to the two axioms of probability, the probability distribution must assign a non-negative probability to each possible outcome, and the sum of the probabilities of all the possible outcomes should be one. The validation of the axioms can be defined as:

```
def sumEqualsOne[T](dist: Dist[T], allOutcomesProbs: List[T]): Boolean =
  allOutcomesProbs.map(dist.probOf).sum == 1.0

def nonNegativeProb[T](dist: Dist[T], allOutcomesProbs: List[T]): Boolean =
   allOutcomesProbs.map(dist.probOf).forall(_ >= 0)
```

In the implementation of the Bernoulli distribution (Listing 5.2), the function probOf corresponds to the probability mass function of the distribution. In discrete distributions it is possible to define a probability for a specific possible outcome. However, in continuous distributions there are infinite values and the probability that a random variable will be exactly equal to a specific value is zero. For example, in the case of a random variable following a continuous uniform distribution in the interval [0, 1], we can only calculate the probability that the variable will take a value between a given interval [a, b] $Pr(a < X \leq b)$, and not a single value from the infinite number of values in the interval. Therefore, for the cases of continuous distributions we would like to evaluate the probability density function, and more specifically the natural logarithm of it, `logpdf`. Respectively, the discrete distributions involve the natural logarithm of the probability mass function `logpmf`. The abstract interfaces that code for the cases of continuous and discrete variables are:

```
trait ContinuousDist[T]{
   def logpdf(t: T): Double
   def generate: Rand[T]
}

trait DiscreteDist[T]{
   def logpmf(t: T): Double
   def generate: Rand[T]
}
```

Listing 5.3: Abstract interfaces for discrete and continuous distributions

We can use both distributions `Dist` and `Rand` to build a function that takes deterministic inputs and returns stochastic outputs as follows:

```
case class Binomial(n: Int, p: Double) extends Dist[Int] with Rand[Double] {
// Invoke probabilityOf() from Breeze library
override def probOf(k: Int): Double =
    breeze.stats.distributions.Binomial(n,p).probabilityOf(k)
override def draw(): Double = breeze.stats.distributions.Binomial(n,p).draw()
}

def createBinomialDist(N: Int, l: Double): Dist[Int] =
  new Dist[Int] {
    def probOf(t: Int) = Binomial(N,l).probOf(t)
  }

def createBinomialRand(N: Int, l: Double): Rand[Double] =
  new Rand[Double] {
    def draw = Binomial(N,l).draw()
  }
```

We can also use the sampling based distribution to approximate the probability of a random variable having a specific value if we use a large number of samples, as we can see in the following example:

```
class ApproxProb[T](rand: Rand[T]){
  val sampleNo = 1000000
  def probOf(t: T) = {
    def calcSuccessCount(sC: Int, i: Int): Int = {
      if (i == sampleNo) sC
      else if (rand.draw == t) calcSuccessCount(sC + 1, i + 1)
      else calcSuccessCount(sC, i + 1)
    }
    calcSuccessCount(0, 0).toDouble / sampleNo.toDouble
  }
}

val randBinom = createBinomialRand(10,0.5)
val approxProb = new ApproxProb[Double](randBinom)
approxProb.probOf(3) // Returns: 0.1173. . .
Binomial(10,0.5).probOf(3) // Returns: 0.1171...
```

The average of a large number of draws will approximate the value of `probOf` of `Dist`. An implementation of a function `sample` should be added to enable us to sample many times from the sampling based distribution.

Developing a functional probabilistic programming language or library is a complex procedure that necessitates combining concepts from category and measure theory, lambda calculus, functional programming, the compilation process and other related fields. By hiding implementation details, probabilistic programming renders probabilistic modelling more accessible to people who do not have sufficient expertise in probability theory but are involved in the field of statistics and machine learning and want to broaden the range of tools they are using for inference.

5.4.2 Rainier – a probabilistic programming library for Scala

Rainier is a probabilistic programming language written in Scala for fitting Bayesian statistical models using MCMC. It is developed by Stripe, an American company that builds software platforms for online financial transactions. Rainier was first available in 2018 and its primary author is Avi Bryant (Bryant, 2018). It supports models with fixed structure and continuous parameters.

As other domain specific languages (DSLs), Rainier associates a compute graph to the predefined model, with the nodes representing the parameters of the model determined by the user. Then, it translates this graph to a function that represents the un-normalised log-posterior density. The library uses HMC and eHMC, hence it involves calculation of the gradient of the log-posterior density since the exact derivatives are necessary in order to perform the leapfrog steps. Many inference and optimisation algorithms use gradients to adjust their parameters in a direction that will minimise the error of a specific task that they aim to accomplish. The wide use of differentiation motivated the scientific community to develop more generic processes and *automatic differentiation* (AD) routines. Automatic differentiation is a set of techniques to calculate the gradient of functions and at the same time numerically evaluate them. It is based on the idea that a computer program, irrespective of its complexity, can be converted to a sequence of primitive operations which determine specified procedures for computing derivatives using the chain rule. AD techniques include "Forward Mode AD" and "Reverse Mode AD" (Wang *et al.*, 2019). Rainier uses AD to compute the gradient of the log-density function and implement the Hamiltonian Monte Carlo algorithm. Developing a model in Rainier requires a level of familiarity with Scala and the basic concepts of functional programming, in order to understand and take advantage of the functional features embedded in this probabilistic language.

To demonstrate how Rainier works in version 0.2.2 we will use a simple one-way Anova example as a starting point in building a probabilistic program with this library. Suppose that the synthetic dataset consists of n groups/levels, with N observations per group. The data is stored in a two-dimensional array with n rows, where each row represents a group, and N columns with the columns representing the observations within the groups. This model in Rainier is defined as follows:

```scala
// Define parameters and priors
val prior = for {
  mu <- Normal(0, 100).param // overall mean
  sdObs <- LogNormal(0, 10).param // observational sd
  sdEff <- LogNormal(1, 5).param // random effect sd
} yield Map("mu" -> mu, "sdObs" -> sdObs, "sdEff" -> sdEff) // save to a Map

// Update by observing the data per group
def addEffect(current: Map[String, Real], i: Int): RandomVariable[Real] =
  for {
    gm <- Normal(0, current("sdEff")).param
    _ <- Normal(current("mu") + gm, current("sdObs")).fit(data(i))
  } yield gm

// Apply to all groups
val model = for {
  current <- prior
  _ <- RandomVariable.traverse((0 until n) map (addEffect(current, _)))
} yield current

// Fit the model
implicit val rng = ScalaRNG(3)
val iterations = 10000
val thin = 1
val lfrogStep = 50
val output = model.sample(HMC(lfrogStep), warmupIterations = 1000, iterations *
    thin, thin)
```

Listing 5.4: One-way Anova in Rainier *(version 0.2.2)*

Rainier has a special numeric type to represent the parameter values of the model, which is called `Real`. It is a random variable and reflects a set of possible values and not a specific known value. Listing 5.4 shows that the first step of a probabilistic model in Rainier is to set up the prior distributions for the unknown parameters of interest. This is defined in the variable `prior`, using the typical Scala for-comprehension (5.3.2.2) and calling `param` on the distribution of the prior for each parameter. The method `param`, allows the distribution to be used as a prior distribution by returning a `RandomVariable[Real]` which is Rainier's probability monad (5.4.1). The library includes and supports various predefined distributions that are used to build the function to evaluate the log of the un-normalised posterior. Using `param` actually transforms the parameters to the support of the distribution. Some distributions, such as the Gamma and the half-Cauchy, have bounded support which needs to be taken into account by transforming the parameters in a way that leads to proposals in an unconstrained space. Therefore there are embedded functions that add the log-Jacobian of the transformation to the log-density and enable an unconstrained random walk proposal (Bryant, 2018).

The second step is to fit the parameters to the observational data and update our beliefs about their values using `fit` (in this example, per group). The result of applying `fit` on a distribution is the enabling of the log-density of that distribution to represent the likelihood of the provided data being produced by that specific distribution. Finally, the last step is to generate samples of the posterior distribution of interest by defining the number of iterations and the thinning factor, as well as the number of leapfrog steps for the HMC. Implementation of a two-way Anova using a random effects model with and without interactions is available at the GitHub repository TWI_Saturated_Rainier_v022 [1].

As of February 2020, a new version of Rainier (0.3.2) is available with significant changes compared to the previous (0.2.2). It is worth showing the evolution of the library by embedding the appropriate alterations in the implementation of the above one-way Anova example, that leads to the following listing 5.5:

```scala
// Define parameters and priors
val mu = Normal(0, 100).latent
val sdObs = LogNormal(0, 10).latent
val sdEff = LogNormal(1, 5).latent

val eff = Vector.fill(n)(Normal(mu, sdEff).latent)

// Update by observing the data per group
```

[1] TWI_Saturated_Rainier_v022: `https://github.com/antoniakon/TWI_Saturated_Rainier_v022`.

```scala
val models = (0 until n).map(i =>
     Model.observe(data(i), Normal(eff(i), sdObs)))

// Merge the different models
val model = models.reduce{(m1, m2) => m1.merge(m2)}

// Fit the model
implicit val rng = ScalaRNG(3)
val iterations = 2500
val lfrogStep = 50
// The new implementantion of HMC runs 4 parallel chains
val output = model.sample(HMC(warmupIterations = 1000, iterations, lfrogStep))
```

Listing 5.5: One-way Anova in Rainier *(version 0.3.2)*

The logic behind the implementation remains the same. First, the user defines the parameters and their prior distributions. Then, the distributions are used to update our beliefs about the random variables of interest, based on the observed data, and finally we run the sampler on the model. However, the details of building the probabilistic model are now different. A new latent random variable is created using `latent`, rather than `param` in a for-comprehension. Then, the likelihood is formed by using `observe` which now enables the connection of the prior with the observations. Next, the separately constructed models for each group are merged into one model using `merge`. Finally we sample the full posterior distribution using `sample`, and HMC or eHMC are now running four different chains in parallel.

Rainier 0.3.2 provides a more user friendly but less functional way to build Bayesian models compared to the previous version, that showed more research interest regarding the hands-on experience acquired from delving into structural details of its implementation. Benchmarking tests on the above two versions and snippets have shown that version 0.3.2 presents poorer performance compared to 0.2.2 by roughly 26% increase in runtime, for an one-way Anova example with 50 groups ($n = 50$) and 250 observations per group ($N = 250$). In both cases HMC of ten 10,000 iterations with 50 leapfrog steps was used. The efficiency decreases further as the complexity of the data and the model increases.

The full code of the examples regarding Rainier, along with implementations for a saturated two-way Anova model, is available at GitHub repository TWI_Saturated_Rainier_v032 [2]. Furthermore, we will refer to Rainier again in Section 7.4.2 as it is considered, among others, as an alternative implementation to a variable selection problem.

[2]TWI_Saturated_Rainier_v032: `https://github.com/antoniakon/TWI_Saturated_Rainier_v032`.

5.4.3 Probabilistic programming and functional programming summary

In this section we presented some of the most important type classes in functional programming. We showed that they are influenced by category theory and how they can be implemented in Scala to abstract over structures and functionalities, providing ad-hoc polymorphism. Widely used type classes, such as monads, enable chaining actions in order to build a function pipeline within the program. Implementations for popular type classes are already defined in libraries for functional programming languages, such as *Cats* for Scala. Programmers can use these libraries and extend the implementations provided to develop solutions that will be more structured, less prone to errors and code repetition, and more agile. We then introduced the concept of probabilistic programming used for statistical modelling. Combination of various scientific fields can lead to the development of domain-specific and stand-alone languages or libraries, that can hide the details of probabilistic modelling, rendering it more accessible to users. Finally, we introduced Rainier, a probabilistic programming library in Scala that uses HMC.

In the following section we will be concerned with parallel programming and how it can affect performance.

5.5 Parallel programming in Scala

Parallel computing is the type of computation where many calculations are performed simultaneously. Efficient parallelism needs support from the programming language and the available libraries, the virtual machine, as in the example of Scala and Java that run on the Java Virtual Machine (JVM), the operating system and the hardware. If all the resources are available a parallel program can run faster than its equivalent sequential version under certain conditions.

5.5.1 Fundamental concepts of parallel computing

In earlier stages of parallel computing the main focus was to increase the frequency of the single processor in the CPU. The most important limitation of this approach was the increased power consumption that raised concerns about financial and environmental impacts. Over the last years, parallel programming architectures became more popular due to the advances in parallel hardware. Computation is divided into smaller sub-problems that can then be executed simultaneously while exploiting the underlying parallel hardware, in order to achieve the optimal speed-up. Parallel programming is very useful to increase performance. However the division of a sequential program into smaller problems can sometimes be difficult or impossible. Assuring correctness in the process can also be challenging.

Nowadays, parallelism can be achieved at various levels, depending on the nature and structure of the program. Considering a computer program as a stream of instructions, the computer can group, reorder and execute non-depending instructions in parallel, a technique which is called instruction-level parallelism. Task, also known as function, parallelism is another form of parallelism where streams of instructions, that are considered as different calculations performed on the same or different datasets, can be executed in parallel. Finally, another type of parallelism is called data parallelism, and takes place when the same calculation is performed in parallel along different processor nodes, each working on a subset of the data (Dongarra *et al.*, 2003).

There is a wide range of parallel hardware architectures that enable parallel computing. Some of the most popular classes of parallel computers include, but are not limited to, multicore processors, symmetric multiprocessors, general purpose graphics processing units, and computer clusters. Multicore processors constitute a technology where on one chip there is a processor with multiple processing units, called cores. When more identical processors (either multicore or single-core) which are on different chips, are combined together, they build a computer system named "symmetric multiprocessors". One of the latest trends in parallel computing is the general purpose graphics processing unit (GPGPU). This unit acts as a co-processor in the system. It is connected on the computer and executes a program whenever this is explicitly requested from the host processor. All three previously mentioned classes have a shared memory. Finally, computer clusters are groups of computers connected by a network, each having its own memory (Kuncak, 2016).

In computer science, a process is an instance of a program. The operating system is responsible for scheduling program execution and multitasking. Processes are assigned small time intervals to run on the CPU. Once their time slice is finished, they are replaced by another. Each process has its own memory area, which is secure since processes can only mutate their own data and interprocess communication is not an easy task. Processes contain multiple units as their building blocks called *threads*. Threads can be explicitly defined by the user and triggered from within the program. Another important property of threads is that they share the same memory, rendering information exchange feasible. In the JVM, each thread has its own program stack, a location in memory to store its local variables and methods, as well as its own program counter, to know the position in its sequence of instructions. Even though the program stack is private to the thread, inter-thread communication is achieved through the *heap* memory, which is the part of the memory that all threads have access to. The most important thread in a Scala program running on the JVM is the one that executes the main method. In serial programming the main thread is the only one that starts when a process is running. In parallel programming however, multiple threads can start from a process and run in parallel as determined by the OS, that takes advantage of the parallel hardware technology of the computer.

Threads are created at the OS kernel level and they are expensive since they have some implementational workload. More modern programming approaches involve creation of tasks from within the program rather than threads directly. Tasks can be considered as objects that represent some work that should be done (Kuncak, 2016). Embedded libraries, such as the *executors* in Java, manage these tasks and the threads on which these tasks will run on various cores.

Another term which is frequently used and often confused with parallelism is *concurrency*. Concurrency occurs when there are two or more tasks running simultaneously (Dongarra *et al.*, 2003), but not necessarily in parallel. More specifically, the CPU is not involved in operations such as read or write from and to the memory, as they happen through a communication system that transfers data between these components known as *bus*. In the meantime, while these kinds of operations take place, the CPU is available to execute other tasks. Concurrency can be achieved even on a single-core CPU. Parallelism, by definition, is achieved when multiple cores or processors are available and used.

With blocks of code running in parallel and multiple threads having access to shared memory, without mechanisms to secure the correctness of commands executed, parallel computing can lead to inconsistencies. To avoid potential problems, the main thread, depending on the sequence of the instructions, will have to wait at various parts of the program for the completion of additional threads started. Once the result is computed, it will be returned and the main thread will continue; this is known as “join”. Furthermore, some threads might change variables in the shared memory, that other threads are processing as well, leading to wrong calculations at one thread and concurrency issues. To overcome this problem the JVM has a mechanism called “synchronisation”, so that only one thread can invoke a code block at a point in time. This block is called a *synchronised block* and constitutes a way through which various threads exchange information. Complex programs and high-level frameworks have nested synchronised blocks to ensure that all threads function properly. The disadvantage of synchronised blocks is that the whole block cannot be executed in parallel with another, with the undesirable outcome that we cannot exploit parallelism for potential expensive operations in the rest of the block. To resolve this difficulty, the JVM provides a more modern “locking” technique. Locks can be used at specific points in a block, so the rest of the block can be executed in parallel (Kuncak, 2016). Finally, for operations causing side-effects, there are concurrent collections that a programmer can use, which handle multiple threads simultaneously without needing deeper knowledge of the collection’s internal representation.

When multiple threads are running, a situation known as *deadlock*, can occur. In this condition, two or more threads compete for the same resources and each might be waiting indefinitely for the other to finish and release the resource. Another issue occurring with competing processes is that with using synchronised blocks the competing resources have no specific order. One solution is to keep an order in the resource acquisition, this is valid when locks are used. Assigning unique identifiers to processes and providing the resources based on the identifier, e.g. prioritising the smallest identifier, could be a potential solution. Concerning possible issues related to accessing and modifying the locations of the shared memory, the JVM has a set of rules called the *Java Memory Model*. One of these rules is that if two threads write to different positions in memory, synchronisation is not necessary. A second rule is applied when a thread calls *join* on another thread. There is a guarantee that the second thread will implement all the writes in memory first before `join` returns, ensuring that the first thread will have all the up-to-date information (Kuncak, 2016).

5.5.2 Parallelism

In this section we will see how parallel programming for two of the main types of parallelism, task and data parallelism, can be achieved in Scala. The first type distributes different execution processes/computations across multiple processing units, whereas the second runs the same computation on distributed data across different nodes.

5.5.2.1 Task parallelism

The idea behind task parallelism is to perform independent computations across multiple nodes. A task in this context, is considered to be an operation that leads to a unique value. Thus, one example of task parallelism could be running operations such as summation and finding the maximum value of one or more arrays, in parallel.

Due to implementational details of the various collections, it is more efficient to apply parallelism on some types than others. For example, even though `List` is a widely used collection, due to its implementation where it consists of `head` and `tail`, searching for the position of the middle of the list, or later concatenating lists in order to obtain parallelism, can be expensive. On the other hand, arrays and trees are more efficient.

Suppose that there are two arrays where we want to apply a function `f` to every element and then get the total sum of the elements. The sequential version of that operation, taking an input array and returning the sum of the transformed elements could be implemented as presented in Listing 5.6. For illustrational purposes a while loop is used instead of the embedded `map` function of Scala.

```scala
def sqrForArraySerial(inputArr: Array[Int], f: Int => Int): Int = {
  val outputArr = new Array[Int](inputArr.length)
  var i = 0
  while (i < inputArr.length) {
    outputArr(i) = f(inputArr(i))
    i += 1
  }
  outputArr.sum
}
// Define input
val inputA = Array(2,3,4,5)
val inputB = Array(5,7,8,9)
// Define function for power of two
val sqr = (x:Int) => x * x
// Function call
val sumSeq = sqrForArraySerial(inputA, sqr) + sqrForArraySerial(inputB, sqr)
//sumSeq: Int = 273
```

Listing 5.6: Sequential implementation of map and sum for two arrays

The parallel version can be then implemented as:

```
def sqrForArrayParallel(inputArrA: Array[Int], inputArrB: Array[Int], f: Int =>
    Int) = {
  parallel(sqrForArraySerial(inputArrA, f),
    sqrForArraySerial(inputArrB, f))
}
// Function call
val sumPar = sqrForArrayParallel(inputA, inputB, sqr)
//sumPar: Unit = () result = 273
```

Listing 5.7: Parallel implementation of map and sum for two arrays

The serial and the parallel implementation return the same results. In the serial code all calculations are executed sequentially. In the parallel version, we invoke two function calls in parallel for the two arrays. This is achieved through `parallel`, a function that uses Scala's trait `Future`. When an operation is executed in parallel its result will not be returned instantly but in an asynchronous way. In some cases we might want to be able to process this result further in following computations. Futures (Scala `Future`) provide a way to perform these operations by representing values that may not yet exist, without blocking the underlying execution threads. When a task/future is completed, subsequent executable code, known as callback, can be called to define the operations to be carried on. Futures are monads, so they can be composed through combinators such as `flatMap` and `filter`. The following implementation for `parallel`, used specifically in Listing 5.7, is a simple example of a `Future`:

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import scala.util.{Failure, Success}

def parallel(taskA: => Int, taskB: =>Int) = {
  val fA = Future { taskA }
  val fB = Future { taskB }

  val result = for {
    r1 <- fA
    r2 <- fB
  } yield (r1 + r2)

  result.onComplete {
    case Success(x) => println(s"\nresult = $x")
    case Failure(e) => e.printStackTrace()
  }
}
```

Listing 5.8: Implementation of parallel using Furure

In this example `parallel` takes two arguments, passed *by name* indicated with `:=>`, necessary to achieve parallelisation since otherwise the program would be executed sequentially by first evaluating the first expression and then the second (Kuncak, 2016). It then creates a `Future` for each task and since futures are monads they can be composed using a for comprehension. This returns a new future named `result`. When this future is completed, either through a value in case of success or an exception in case of failure, it applies the provided function. Then, it prints either the result or an exception error. This is an impure operation helpful for illustrative purposes.

Parallelising operations on collections can lead to significant speed-ups, however not all operations are parallelisable. For example, some common operations that lead to single values are `foldLeft/foldRight` and `reduceLeft/reduceRight`. Some examples of how they operate are:

```
Array(1,2,3,4).foldLeft(5)(_-_) //res0: Int = -5
Array(1,2,3,4).foldRight(5)(_-_) //res1: Int = 3
Array(1,2,3,4).reduceLeft(_-_) //res2: Int = -8
Array(1,2,3,4).reduceRight(_-_) //res4: Int = -2
// Signatures
def foldLeft[B](z: B)(f: (B, A) => B): B //similarly for foldRight
def reduceLeft[B >: A](f: (B, A) => B): B //similarly for reduceRight
```

Listing 5.9: fold and reduce examples

In the above examples, `foldLeft` starts from the initial element 5 and applies, from left to right, a function `f` passed as an argument. Here, `f` is a subtraction, so `foldLeft` subtracts the elements from the previous result as: ((5-1)-2)-3)-4). Likewise, `foldRight` applies `f` again, but starts from the last element: (1-(2-(3-(4-5)). `reduce` works in a similar way but without having initial values. Operating on the same array, `foldLeft` and `foldRight` perform the operation `f: (B, A)=> B` in a different order and return different results. Therefore, these operations could not be implemented in parallel and guarantee validity of results. This is due to the nature of the function. In order to perform these operations in parallel and assure correctness of results, function `f` used in these operations must be associative. The associative law for binary operations defines that an operation *f: (A,A)* $\Rightarrow$ *A* is associative iff for every x, y, z: $f(x,f(y,z)) = f(f(x,y),z)$. Associativity allows the reordering of the sequence of computations, that lead to the same value, given that the ordering of elements is preserved. It enables parallel execution and reduction on the parts of the collection, and their combination to a final resulting value.

Another assumption that needs to hold for `foldLeft` and `foldRight` to be parallelisable is that the accumulator type `(z: B)` must be of the same type as the elements in the collection. Even though in this example an integer accumulator was used on an array of integers, we can see from the signature that using an accumulator of a different type from the collection's elements is allowed. This leads to the definition of `fold` with signature: `def fold(z: A)(f: (A, A)=>A): A`. Now, `f` takes two arguments, the accumulator and the current element of the collection, both having the same type, combines them according to the definition of the function and returns an element of the same type. The same applies to `reduce` (Kuncak, 2016).

In the following part we will see how data and collections can be split on several nodes to run computations on them in parallel and return new transformed collections. Operations that we saw in this part, such as `fold`, can then be operated in parallel as well, on the resulting collections.

5.5.2.2 Data parallelism

In data parallelism, the same computation is executed in parallel on different parts of the data, spread across multiple computational units. Operations such as parallel loops belong in this category and can be highly performant compared to their serial implementations. However, when there are side effects we need to ensure that results are written to separate memory locations or use some form of synchronisation, so as to return correct results.

Modern programming languages offer parallel alternatives of their built-in collections. A method `par` is applicable to most collection types and converts them to their equivalent parallel alternatives. The parallel implementations are prefixed with "Par", e.g. `Array` and `ParArray`. Fragments of the structure are then processed in parallel through usual operation calls. Similarly, accessor built-in operations, such as `fold` and `count`, as well as transformer operations, such as `filter` and `map`, can take place in parallel. Nevertheless, not all collections and operations are parallelisable. Parallel collections use embedded language features to split their data, do the independent parallel calculations and combine the separate results.

A simple example of finding the sum of an array using its parallel collection and `fold` is the following:

```
def mySum(ar: Array[Int]): Int = {
  ar.par.fold(0)(_+_)
}
val inputA = Array(2,3,4,5)
mySum(inputA) //res0: Int = 14
```

Listing 5.10: Example of parallel array sum

Here, `par` copies all the elements in the array, in linear time, and creates a new parallel collection. The returned result is correct and the method is deterministic in the sense that it will always return the same output for the same input on different runs. This is valid because two requirements are met. Firstly, the parallel reduction operator, sum, is associative. Secondly, the neutral element not only has the same type as the collection, but also it is the identity element. These properties, make the neutral element `z` and the binary operation `f` form a *monoid*, a term that we also saw in Section 5.3.2.2 when we presented the basic type classes in functional programming. These prerequisites show that there are different aspects to consider when writing parallel programs that may not be applicable in the serial case.

The set of data structures in Scala constitutes a network of collections and subcollections forming a hierarchical system. This applies to both serial and parallel collections. To allow the development of code which works with both categories, there are generic collection traits which are supertypes of both versions. For example, the generic `GenMap` is a supertype for both the serial `Map` and the parallel `ParMap`. This facilitates the programmer in writing more generic and flexible code that will be called either on the serial or on the parallel version of a collection, depending on the nature of the problem to which it is applied (Kuncak, 2016).

5.5.3 Summary of the parallel programming framework

Parallelism is a type of computing that allows the running of multiple threads in parallel on different computing nodes in order to execute operations faster than their serial counterparts. Performance improvement is the main motivation behind parallel programming, that can otherwise be more complex. Especially when dealing with mutable structures and side effects, parallel programming introduces new aspects, like concurrency and inconsistencies, that have to be considered. Luckily there are mechanisms to overcome these problems, such as synchronisation, locking, and built-in concurrent collections. Furthermore, the lack of shared mutable states in functional programming helps to avoid concurrency issues and makes parallelism easier. The two main types of parallelism, task and data parallelism can be applied depending on the problem. The first focuses on running different processes in parallel on the same or different data. Data parallelism aims to distribute data across computing nodes and execute the same computation in parallel. Parallel collections and their supported operations, such as `map` and `fold`, render parallelism an easier task for the programmer, with their underlying implementations handling efficient split, processing and recombination of the data. Often in the literature and online sources there is a very fine line of distinction between task and data parallelism examples. Despite the complex concepts involved in parallel programming, its performance gain renders it very popular for large-scale problems.

5.6 Benchmarking

Benchmarking is a concept used in computing, related to testing and comparing the performance of software, hardware or networks. For programming, it refers to the act of running a computer program in order to assess the performance of the whole system or some of its components. There are various performance metrics, such as runtime, memory utilisation and disk operations and usage.

Measuring runtime is the most common performance metric. It is also the metric that we are going to use in the scope of this project. In order for the measurements to be consistent, not only among various trials on the same program, but also among different programs, some factors affecting runtime must remain fixed during all experiments. More specifically, these factors can be related to the CPU, such as the processor's speed, the number of available processors (the last applies mostly on parallel programming), and the throughput, which measures how many units of information a system can process in a time unit (Kuncak, 2016). They can also be related to memory, and particularly latency and bandwidth. Both terms are associated with retrieving data from the main memory. The former refers to the waiting time from the moment a processor initiates a request for data from memory until it receives them. The latter, regards the rate at which data can actually be read from and stored to memory per time unit (Dongarra *et al.*, 2003). Additionally, another factor that can affect performance is the runtime environment, such as the operating system or a virtual machine. Each system has its own characteristics and components running in the background. For example, the JVM runs a process which is called "Garbage Collection" (GC) every time memory space becomes limited. It is an automated process for cleaning the memory from objects, created during the execution of programs, that are no longer useful. Finally, other programs running at the same time with our program of interest might affect its runtime due to competition in accessing memory and other resources.

To ensure that the runtime recorded reflects the actual speed of the program and gives us a sensible idea of the program's overall behaviour, it is sometimes important, depending on the case, to repeat the measurement multiple times. It is then useful to compute the mean and variance of these measurements, after identifying potential outliers related to the underlying system's implementation, such as the garbage collection mentioned earlier. This is mostly important in applications where the functions of interest are short and the runtime small. Then the impact of GC and lack of a warm-up period can have a notable effect on the overall time, leading to significant inconsistencies between measurements. The easiest way to measure the runtime is to calculate the difference in the system time between the moment that the program of interest started and when it ended. There are also more sophisticated tools and frameworks that offer more integrated solutions to calculating performance metrics. For example, in Scala there is a library called ScalaMeter specifically designed for performance tests and micro-benchmarking. Finally, it is worth clarifying a concept that is sometimes confused with benchmarking which is called profiling. Whereas benchmarking aims to present the overall performance of a program, profiling is about understanding how the time is spent in specific functions or sets of instructions, and eventually optimising and improving the application (Kuncak, 2016).

Chapter 6

Implementing Bayesian hierarchical modelling

Since the beginning of the BUGS (Bayesian inference Using Gibbs Sampling) project in 1989, there has been a significant growth in the application of Bayesian methodologies for both academic and commercial purposes (Lunn *et al.*, 2009). Various statistical packages such as WinBUGS, that runs on Windows, and OpenBUGS, that constitutes its open source equivalent, are widely used for Bayesian modelling. Another piece of software which is also widespread for Bayesian inference is called JAGS (Just Another Gibbs Sampler) and is based on the BUGS language. It is implemented in C++ and is cross-platform. JAGS interfaces well with R through 'rjags', an R package included in the R Archive Network (CRAN) which makes it popular among R users involved in Bayesian statistics. Both BUGS and JAGS use their own language for the textual specification of models that can be built with MCMC algorithms. These packages are established, acknowledged and highly appreciated by the academic and professional communities.

Over the last few years, functional programming has became a popular programming paradigm for many applications, including its usage for probabilistic programming. Therefore, it is interesting to explore a functional approach to Bayesian hierarchical modelling. It is worth investigating whether with functional programming there is potential for improvement in speed and performance issues that might arise due to the increasing volume of data and complexity of models. However, unlike R, where there are libraries that enable users to define at a higher level the structure and parameters of models and use a Gibbs sampler, the particular MCMC algorithm that we are interested in, to approximate the values of interest, in Scala there is no equivalent active library at the time of this project. So, in order to build a model using a Gibbs sampler in Scala it is necessary to analytically implement the algorithm after obtaining the analytical forms of the full conditional distributions of each parameter to sample from at every iteration, as shown in Section 3.3, and code up the sampling process.

The main model that we examine in this project is an extension of a hierarchical two-way Anova problem. It involves two independent categorical variables and identification of the most important interaction terms through variable selection. All effects are treated as random effects. It also accounts for four cases of asymmetry in the effects according to the analysis presented in Section 3.3, that is, the main and/or interaction effects being symmetric/asymmetric. For variable selection we consider two main approaches, the Kuo and Mallick method using variable selection indicators, and the use of adaptive shrinkage priors, and more precisely, the Horseshoe prior. For both methodologies a Gibbs sampler in Scala and JAGS was developed and all variations of asymmetry in the effects were considered. In addition, since adaptive shrinkage priors are mostly intended to be used for variable selection in algorithms that do not support discrete variables, such as Hamiltonian Monte Carlo, two supplementary implementations were built. The two libraries explored to showcase how HMC is combined with the Horseshoe are Stan for R and Rainier for Scala. These two programs only implement the asymmetric main and interaction effects case.

In this chapter we present some implementational details of the alternative probabilistic programming approaches considered, with particular emphasis on the development of the code in Scala which will be referred to as TWIiS (Two-way Interactions in Scala) indicating its main purpose. For TWIiS, along with a serial approach, a parallel implementation is also examined and a comparison between the two is performed to clarify under which conditions each version is preferable.

6.1 JAGS

Developing code in JAGS has to follow the norms of the language, in order for it to compile and produce useful output. The main points of the JAGS workflow include definition of the model and compilation, initialisation of the parameter values, adaptation and run of the model, and finally monitoring the parameters of interest (Plummer, 2017). Our program uses the library 'rjags' and combines R and JAGS to build the examined model. The code follows a typical pattern. It starts with reading and preprocessing the data, and setting the initial values of the model. Next, it follows the definition of the model in JAGS based on the rules of the language. The stochastic and deterministic relations among parameters are clearly defined for all four cases of asymmetry in the effects. Depending on how we set the logical flags defining the existence of symmetries, a different part of the code is triggered, gradually constructing and eventually leading to a complete definition of the model. Then, the model is set to run, after specifying characteristics such as the number of chains and iterations, the size of burn-in, and the parameters to be monitored. Internally, a graphical model is constructed and the conditional dependencies are used to produce the full conditionals of all variables, followed by the application of a Gibbs sampler. Finally, the saved posterior samples are processed using R. The meta-analysis includes production of diagnostic graphs and tests for convergence, as mentioned in Section 2.2.4, as well as measures of fit and efficiency. This process is typical in statistical modelling using R and JAGS. Numerous variations can occur depending on the nature of the model and the data. An example is the implementation of variable selection using Horseshoe priors instead of selection indicators, even though the two methodologies have some important conceptual differences, only some minor changes in the JAGS code were necessary. Overall, the consensus of defining the modelling process remains similar for most Bayesian models fitted in JAGS.

6.2 TWIiS

Unlike R, where there are libraries to facilitate Bayesian inference, developing a Gibbs sampler in Scala is more challenging since at the time of this research project there was no active library serving a similar purpose. Therefore, a lot of time and work was allocated to develop from the ground up this MCMC algorithm implementation in Scala, focused on our model of interest. The aim of the following sections is to unfold the flow of the code and explain its major components. The building blocks of the program combine the framework of the algorithm established in Section 2.2.2, the calculations of the full conditional distributions presented in Section 3.3 and the principles and techniques for functional programming in Scala discussed in Chapter 5. Finally, we are going to expand the serial version and see how parallelism can be incorporated in the code. The code for developing TWIiS using variable selection indicators is available at the GitHub repository TWIiS_VSI [1], and with the Horseshoe priors at repository TWIiS_HS [2].

6.2.1 Main structure and optimisation

One of the main parts during the development of the code was the decision upon the implementation of the basic structure that would hold and manipulate the data. In this part we will show why the form of this structure is important and mention some of the implementations tested.

The nature of our hierarchical model requires estimation of several explanatory variables using their full conditional distributions, leading to numerous calculations. There are also four different variations of the model depending on the symmetry of the effects to be taken under consideration. It was important to identify the structure that best fitted the

[1]TWIiS_VSI: `https://github.com/antoniakon/TWIiS_VSI`.
[2]TWIiS_HS: `https://github.com/antoniakon/TWIiS_HS`.

computational and speed requirements of the model, therefore a lot of trials took place. An easy way to be able to experiment on various implementations is to define a *trait* (also known as *interface* in other programming languages) that encapsulates methods, that will then be defined and implemented by the classes that extend it. The classes of structures tested work in this manner. After defining the trait, for every different class that extends that trait and holds a different implementation of the structure, it is necessary to define exactly how the methods work to perform their functionalities.

The structure of the dataset is simple; for each observation there is information about its response value and the level of each of the two categorical variables. Most synthetic datasets have 10,000 observations and the real datasets have approximately 80,000 observations, resulting to a large number of calculations during the sampling process. When updating the parameters of interest using their full conditional distributions there are a lot of selections of subsets of data and numerous calculations performed on these sets. For example, in the case of two main effects when updating a specific first main effect all the observations that include it must be selected and summed. Furthermore, all the effects of the second main effect for which the observation has the first effect equal to the effect being updated have to be accounted for and summed over. In addition, there are summations over the interaction effects involving the main effect being updated. These selections and calculations get more perplexing when there is symmetry in the effects involved. For instance, when updating a main symmetric effect, it is necessary to identify and select all the observations for which this effect is either in the first or the second categorical variable and then perform the necessary calculations on this group. Consequently, there is a need to store the data in a structure that will enable easy access to specific combinations of main effects and allow for fast main operations, such as summations and counts. Structures that loop over all their data to search, find and retrieve data can cause delays and inefficiencies.

For a serial implementation of the Gibbs sampler in Scala, the simplest architecture to store and access the data which was tested was a two-dimensional array of lists. Each dimension represents each one of the two categorical variables and the list holds the observations of the group. Even though a multidimensional array offers a straightforward way to access specific data, the major problem that appeared during the trial on a synthetic dataset was the fact that this implementation lacks support for "missing" levels of either one of the categorical variables. Serial access and looping through the structure and multidimensional array leads to "index out of bound" exceptions when an array is empty. Possible workarounds could solve the problem, but potential bugs and errors in calculations would then be difficult to debug even in smaller synthetic datasets.

To overcome the problem with the implementation that used arrays, we developed an implementation of a structure that used *Maps*. Maps are dictionary-like containers offering the possibility to use keys of some form for faster lookup. In our example the keys are a pair of integers representing the two levels of the categorical variables. Each key has a custom list that contains the corresponding observations of the group. Using maps allows accessing the data for each key in the map and since the keys are created from the data, they will only contain levels of categorical variables that are present in the actual dataset. Hence, we secure that "missing" levels will not interrupt the flow of the code.

Additional substructures in the structure class, in later stages of the development, aimed in further optimisation of the serial version by enabling preprocessing and storage of repeated calculations over each iteration. Another programming feature that was applied in the methods of the structure objects and proved useful in moving towards performance improvement is called *memoisation*. It is an optimisation technique for expensive functions and can be applied irrespective of the collection used. The first time that an expensive function is called its result is stored and in subsequent function calls with the same input it is returned directly without being computed again. Further techniques for optimisation will be presented in Section 6.2.3.

To conclude, the type of structure plays a key role in the efficiency and speed of the methods that act on the data to perform calculations over specific groups, observations and effects.

6.2.2 Program flow

A convenient way to illustrate the flow of TWIiS is through a graphical representation of the steps followed, as shown in Figure 6.1. The description that follows pertains to both the implementation using variable selection indicators and the one using Horseshoe priors. There are two main implementational differences between the two approaches. The first is the step to update the inclusion probability which applies only to the variable selection indicators method. The second difference is associated with the stage that updates the interaction coefficients and will be further clarified when this step is demonstrated below. The basic processes that compose the program are:

- The program starts by reading the input data file in Comma Separated Values (CSV) form. In our examples this was either a synthetic dataset or one of the multiple real data files coming from the yeast genome case study examined. In either case the data were clean and did not contain any missing values.
- *Preprocessing Stage 1* is related to processing the data. At this point the number of levels for each categorical variable is extracted and any prior assumptions about the parameters of the model are defined. Furthermore, there is a call to the constructors of the classes that create the structures using maps to store the data, as mentioned in Section 6.2.1.

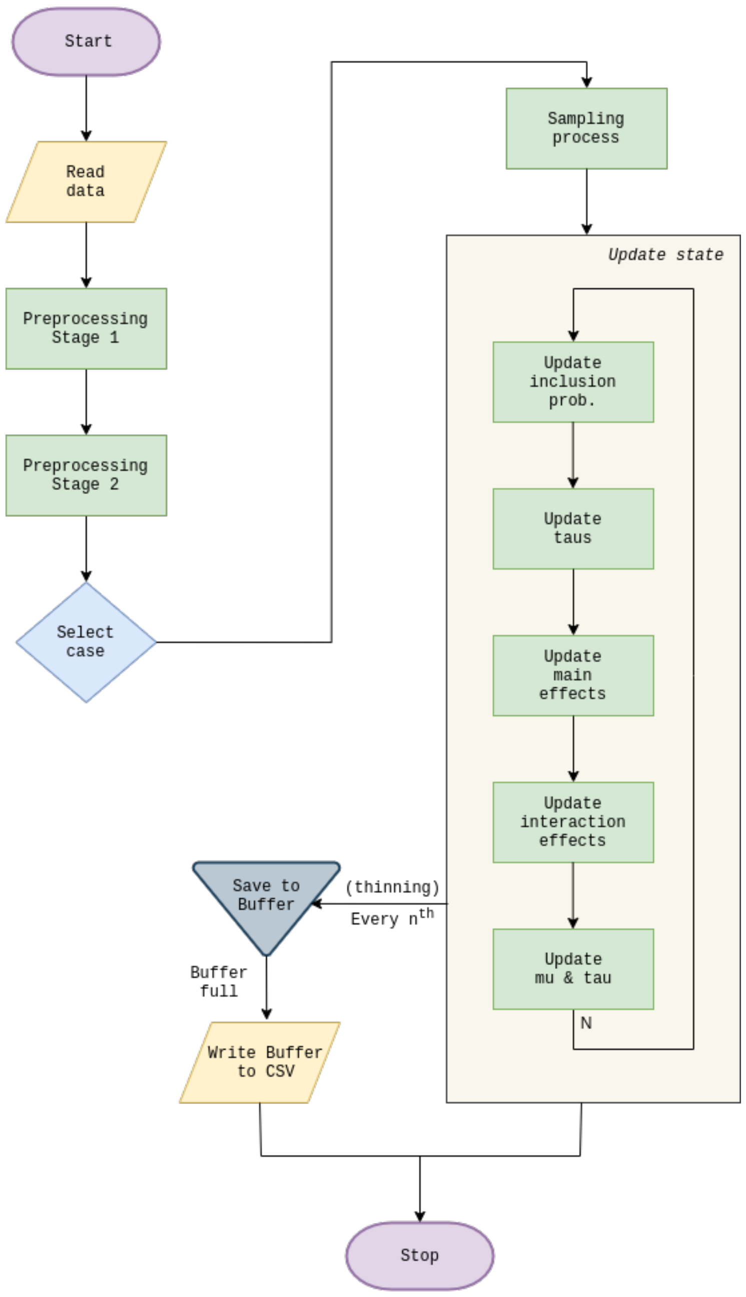


Figure 6.1: Scala code flowchart

There are two main structures created, the first has as keys pairs of the levels of the categorical variables unsorted, for example (CV1, CV2) where the first term represents the level of the first categorical variable and the second term the level of the second. The second structure has the levels sorted (min(CV1, CV2), max(CV1, CV2)), irrespective of which level belongs to which variable. The latter is useful in the implementation of symmetric cases. The fact that in this structure there is no knowledge about the order of the levels is in accordance with the rationale of symmetry in interaction effects where the order is of no importance.

- *Preprocessing Stage 2* is the part of the program where the structures are actually created and their methods defined. In the related class, apart from the creation of the main structures from the data, there are also various other private structures created that aim to allow methods to perform computations faster since they have the data already gathered in useful forms. Some computationally demanding calculations that are repeated at every iteration of the algorithm but remain unchanged, e.g. summations of group observations, are pre-calculated at this point and stored in memory to accelerate the sampling process. This tactic created no memory overflows with the volume of data involved in our models.
- The stage *Select case* is the point where the user has defined which one of the four possible models he/she wants to run, the equivalent object is created and the method triggering the sampling process is called on that object.
- The *Sampling process* runs N times, where N is the number of iterations. At this stage, each unknown variable is sampled using its full conditional. As defined by the Gibbs sampler algorithm, we start with a state of the unknown variables (usually initialised at zero or one) and at the end of each iteration, after having updated each variable, we get the fully updated state which is then used for sampling the parameters at the next iteration. For each parameter that has already been updated at the current iteration and is necessary for updating another parameter, the recently updated value is used.

  For our example, it appeared reasonable to divide the unknown variables into four groups, consequently the sampling process logically and implementationally consists of four stages:

  1. *Update inclusion probability* is associated with the process and equivalent method that updates the inclusion probability (p) of the interaction effects and defines the sparsity of the model. It is implemented only for the variable selection indicators approach.
  2. *Update τs* represents the process and corresponding method that updates the precisions (τ) of the main and interaction effects. Please note that in the version implementing the Horseshoe prior there is no precision for the interaction effects.
  3. The main effects are updated at this stage. Depending on the case, either symmetric or asymmetric main effects, each object reflecting the case has a different

implementation of this function, to incorporate the differences in calculations as defined by the full conditionals.

4. This step involves updating the terms for the interaction effects. As in the previous step, each object has its own implementation associated with the existence of symmetry or not.

   At this point it is worth clarifying that this is the step where there is a major implementational difference between the version using variable selection indicators to identify the important interactions and the version using Horseshoe priors. This difference is based on the logic behind the two methodologies and the corresponding derivation of the full conditionals of the involved parameters. In the former method, there is an update for each interaction coefficient regarding its strength and an update for its equivalent variable selection indicator. The final coefficient is their product. In the latter method, there is a common variable τ_{HS} for all the interaction coefficients that causes shrinkage to zero, and a variable λ associated with each coefficient separately that allows some coefficients to escape shrinkage. All these parameters are estimated at each state using high dimensional matrices and combined calculations. In the case of the Horseshoe prior, automatic adaptation of the tuning parameters, necessary for the correct and efficient function of the algorithm, is also included. It is based on the "Adaptive Metropolis–within–Gibbs" algorithm proposed by Roberts and Rosenthal (2009), where during the burn-in period, an increment is added to or removed from auxiliary variables, adapted according to the acceptance ratio.

5. The final step of the sampling process is the update of the mean (μ) and precision (τ). If the log-likelihood needs to be calculated, it takes place at this stage. This group of variables is not randomly chosen to be the last group updated at each iteration. This decision is based upon the fact that the calculation of the log-likelihood, that must happen at the fully updated state, has some common mathematical operations with the calculation of the rate parameter in the Gamma distribution from which τ is sampled. Hence, we avoid calculating the same values multiple times, which is an important factor considering the numerous iterations required.

- At the same time as the sampling process, a CSV file is opened to save the output every time the buffer is full. However, not all states produced from the sampling process are stored in the buffer. Only every n^{th} element, with n representing the number of the thinning factor, is saved to the output file.

- The buffer is set to have a storage capacity of 1,000 states, where each state consists of the updated parameter vector holding all the recently updated variables. When the buffer is full, it writes its content to the CSV and is emptied.

- The program finishes when all iterations are completed and when the last buffered results are saved to the output CSV.

6.2.3 Parallel implementation

In Section 5.5 we distinguished between two main types of parallelism, task and data parallelism. We mentioned that the first category mainly refers to running different computations on multiple cores on the same or different datasets, whereas the second is the simultaneous execution of the same computation in parallel across the elements of a dataset. In this part we will explore if and how parallelism techniques can be used in our program, and test whether they result in a considerable speed-up.

The Flowchart 6.1 illustrated the main logical and implementational parts of TWIiS, our implementation of a Gibbs sampler in Scala that aims to perform variable selection on the interactions of a two-way Anova model. Looking at the clear demarcation of the tasks and functions, parallelisation through an adaptation of the example in Listing 5.7, following the abstract logic:

```
def updateState(list of arguments ...): newState = {
  parallel(updateTaus(...), updateMainEffects(...),
      updateInteractionEffects(...), updateMuTau(...))
}
// block of code to call updateState using recursion or a for-loop
{
   ⋮
   updateState(...)
}
```

would seem a reasonable solution. However, thinking about the algorithmic steps of the Gibbs sampler that impose dependence of groups of variables on the latest updated values of others makes it clear this type of parallelism is not feasible, at least not across different update functions as only conditionally independent variables can be computed at the same time. This is also the reason why, in within-function level in the case of symmetric main effects, we cannot consider parallelising the updates of the main effects zs as they depend not only on other values, such as τ_z, and μ, but also on other zs. On the other hand, we can consider using parallelism on the asymmetric main effects cases, on each main effect separately. The main effects αs are conditionally independent; each α only depends on other variables, such as μ and τ, but not on other αs. Similarly, updating βs and the interaction effects can also be parallelised. Therefore, αs, βs and interaction effects could be updated in parallel in their own update function. In addition, some summations in various parts of the program can also be calculated in parallel, contributing to further acceleration of runtime.

Sometimes, the overhead of using parallelisation, related to the extra amount of work in creating and deleting threads as well as splitting and combining results, can be greater than the actual benefit of parallelism. This can lead to either insignificant speed-ups or even delays when the data or collection on which parallelisation is applied, is not big enough. There is no golden rule for the size of a collection, so that parallelism would really have a positive effect. Usually, experimentation with various techniques on different parts of the program provides an insight about the benefit or not of using parallel approaches.

In our program, firstly we focused on updating αs in parallel and βs in parallel, which did not lead to improved runtimes. This is most probably due to the "small" size of an internal helper `Map` structure, different from the main structure described in 6.2.1. This `Map`, used to update the main effects, has 113 keys in the case of the biggest example on which we run the program. Therefore, for the main effects we kept the sequential update, to avoid the extra complexity of accounting for concurrency problems when accessing the same locations in memory.

Our next aim was to apply parallelism on the interactions update. Since the number of interactions is the product of the number of levels of the two main effects resulting in a collection of a significant size, parallelisation, theoretically, should have a positive impact on speed-up. Both the serial and the parallel versions, the code of which at an abstract level will be presented below, implement the function `nextIndicsInters` which updates the indicators and the interaction coefficients, and calculates their product as well. It takes two case class instances as arguments. The first, `oldfullState`, contains the updates for the unknown parameters estimated at the previous step of the algorithm. The second, `info`, contains general information such as priors, the number of iterations, the structures and the number of levels for the first and the second categorical variable, denoted as `info.alphaLevels` and `info.betaLevels` respectively. In both versions there are two matrices for storing the results of the indicators and the interaction coefficients, `curIndicsEstim` and `curThetaEstim` of type `DenseMatrix`, which is a matrix implementation provided by Breeze (2020), a library for numerical processing.

The serial version (Listing 6.1) for updating the interactions starts with declaring an initialised with zeros, mutable matrix of size $M \times N$, where M is the number of levels for the first main effect and N for the second. Then, we access each element, representing a pair of interactions, of the main `Map` structure (Section 6.2.1), using `foreach`. We apply the necessary calculations to get the new coefficients and save the result to the correct position in the matrix. After exhausting the whole structure, at the end of the step, the matrix holds all the recently updated estimates of the existing interactions.

```
override def nextIndicsInters(oldfullState: FullState, info: InitialInfo):
    FullState = {
  val curIndicsEstim = DenseMatrix.zeros[Double](info.alphaLevels,
      info.betaLevels)
  val curThetaEstim = DenseMatrix.zeros[Double](info.alphaLevels,
      info.betaLevels)

// The type of structure is a Map[(Int, Int), DVList]}
  info.structure.foreach(item => {
    // block of code for calculations to estimate current indicator and theta
         ⋮
    curIndicsEstim(item.a, item.b) = ... // 0 or 1
    curThetaEstim(item.a, item.b) = breeze.stats.distributions.Gaussian(mean,
        sqrt(1 / tauInter)).draw()
  })
  oldfullState.copy(thcoefs = curThetaEstim, indics = curIndicsEstim,
      finalCoefs = curThetaEstim *:* curIndicsEstim)
}
```

Listing 6.1: Example of sequential update of interactions

Alternatively, a more functional solution could be to use `map` instead of `foreach`. Then, we would not have side effects of mutating the matrix inside a loop. However, there was a performance cost which led to adopting the non-functional approach.

For the parallel implementation (Listing 6.2) we turn the `Map` structure into a parallel `Map` (`parMap`). We then apply the interaction update function on the parallel structure. This means that all the elements are copied to a parallel collection that recursively splits the collection, applies the operation on each partition in parallel and recombines the results. There could be a considerable speed-up since the same operation is invoked in parallel on different partitions of the data. However, we have not yet accounted for potential concurrency problems occurring when multiple threads write on memory at the same time by accessing simultaneously the matrix holding the results. This requires the `map` instead of `foreach` approach described above to avoid possible concurrency problems. Hence, the resulting collection is again a parallel `Map` as our data structure. We then populate the matrix where we save the results by sequentially accessing the resulting parallel `ParMap`. By using `seq` we ensure that the results are consistent.

```
override def nextIndicsInters(oldfullState: FullState, info: InitialInfo):
    FullState = {
  val curIndicsEstim = DenseMatrix.zeros[Double](info.alphaLevels,
      info.betaLevels)
  val curThetaEstim = DenseMatrix.zeros[Double](info.alphaLevels,
      info.betaLevels)

   //The type of structure is a ParMap[(Int, Int), DVList]}
   val estimations = info.structure.map(item => ((item.a, item.b), {
     // block of code for calculations to estimate current indicator and theta
          ⋮
     val indicsEstim = ... // 0 or 1
     val thetaEstim = breeze.stats.distributions.Gaussian(mean, sqrt(1 /
         tauInter)).draw()
     (indicsEstim, thetaEstim)
    }))

   estimations.seq //make sure we are on single thread to access the collections
       above without concurrency problem
     .foreach { case (key, value) =>
       curIndicsEstim(key._1, key._2) = value._1
       curThetaEstim(key._1, key._2) = value._2
       }

   oldfullState.copy(thcoefs = curThetaEstim, indics = curIndicsEstim,
       finalCoefs = curThetaEstim *:* curIndicsEstim)
}
```

Listing 6.2: Example of parallel update of interactions

We confirm the correctness of the parallel implementation by running the program multiple times and comparing the results of each run to the results derived from the serial version. For the comparison we use the synthetic datasets of the simulation study and the posterior analysis methods that are presented in Section 7.2 of the following chapter.

Another remark concerning the parallel implementation is the need to ensure that the random number generator, used when sampling from a distribution, supports concurrent applications. Here we are using the Breeze package. Earlier versions of the library created identical random number generators for each thread, and consequently the same random numbers. Later versions of the library support the parallel mode. If multiple threads use the random number generator, each thread gets a new generator with a different state.

Finally, it is worth mentioning that Scala is not the only language that provides programming tools such as the parallel collections that we examined above and used for the parallel version of TWIiS. For example, Java SE 9 introduced streams which make functional manipulation of collections possible (e.g. using map and fold). Part of this streams feature is a method called *parallelStream* which returns a stream that will process the collection in a parallel way similarly to the parallel collections in Scala.

In the next part we will test the impact of parallelism on the runtime of TWIiS and compare it to the serial implementation.

6.2.4 Efficiency comparison parallel vs serial implementation

Here, we are going to explore the impact of parallelism on the performance of TWIiS. More specifically we are going to compare the parallel implementation that uses data parallelism and a parallel Map for the main structure, as discussed earlier, with two serial versions. In order to make these comparisons we will first introduce the concept of speed-up in parallel computing, as well as two important laws associated with the theoretical maximum speed-up that can be achieved through parallelism.

In parallel computing, speed-up or speed-up ratio measures the relative performance of the serial and the equivalent parallel system. It is calculated typically as:

$$\text{speed-up} = T_1/T_N,$$

where T_1 denotes the computational time of the serial version, and T_N is the computational time of running the parallel version of a program using N processors. The speed-up ratio shows how much faster the parallel version is compared to the serial.

In an ideal situation, if T_1 is the computational time of the serial version and there are N processors available, the parallel time T_N would be T_1/N and there would be a linear relationship between the speed-up and the number of processors. However, not all parts of a program can be parallelised. In parallel computing, there are two main laws, Amdahl's law and Gustafson's law, developed in 1980s, that account for this limitation of the serial part that is not parallelisable. According to Gustaphson (1988), they both define the maximum theoretical speed-up using fractions of times spent on strictly sequential parts and on parts that can be executed in parallel. Amdahl's law is defined as:

$$\text{speed-up}_{\text{AmdahlLaw}} = 1/(s + p/N), \tag{6.1}$$

where s is the proportion of the runtime spent on the serial part and p is the proportion of the runtime spent on the part that is parallelisable. According to Amdahl's formula, speed-up has an upper limit that depends on the proportion of execution time that the serial part holds. Amdahl's law sets a theoretical background for the potential speed-up in a fixed size problem.

On the other hand, Gustafson's law focuses on the idea that as the number of available resources increases, people tend to increase the size of problems to exploit the computing power provided, against the assumption of fixed workload of Amdahl's law. The maximum speed-up of large problems according to this law is defined as:

$$\text{speed-up}_{\text{GustafsonLaw}} = s + p \cdot N. \tag{6.2}$$

The notation for Gustafson's law is the same as for Amdahl's law. However, here, N can be perceived not only as strictly the number of processors, but also as an indirect measure of the problem size, since increasing the number of available processors, theoretically, allows an equivalent increase in the amount of work executed in parallel in the same amount of time. As a final remark about these laws, we should mention that neither of the two accounts for memory limitations that can bound the problem size and the potential speed-up. Subsequent laws, such as Sun–Ni's Law (Sun and Ni, 1990), incorporate the memory capacity in their estimation of theoretical speed-up.

In order to explore the impact of parallelism in speed-up we considered two serial implementations. The first serial case is the fastest serial version achieved and includes techniques to speed up the code, such as mutable collections, that can lead to concurrency problems when used in the parallel case. Therefore, it cannot be directly "translated" to parallel implementation without modifications. The second serial version is in exact accordance with the parallel, in the sense that it is the same code as the parallel version, with the difference that instead of using a parallel Map, we are using a regular Map. In other words, the former serial version is not concurrent safe to be used in parallel mode. However, it is faster than the latter which is more functional and can easily and safely be transformed to a parallel implementation.

Table 6.1: Efficiency comparison between serial and parallel.
Runtimes (RT) are measured in seconds.

Example	Version	RT	Speed-up	RT	Speed-up	RT	Speed-up
		10k obs.		80k obs.		300k obs.	
15×20 (100k iter.)	Serial	45s	0.32	210s	1.09	985s	2.70
	Serial-conc. safe	65s	0.47	339s	1.77	1158s	3.17
	Parallel	139s	-	192s	-	365s	-
		80k obs.		200k obs.		500k obs.	
113×143 (10k iter.)	Serial	126s	1.40	181s	1.77	342s	2.39
	Serial-conc. safe	143s	1.59	285s	2.79	419s	2.93
	Parallel	90s	-	102s	-	143s	-

The comparisons among the three versions are illustrated in Table 6.1 above. We denote as "Serial" the faster, less functional serial implementation. The concurrent safe serial approach is referred to as “Serial-conc. safe”, and the parallel is denoted as “Parallel”. We are interested in not only measuring the time and speed-up of the three implementations but also in understanding if and how the volume of data affects the performance. Therefore, we use synthetic datasets with varying sizes. The first example has 15 levels for the first and 20 levels for the second categorical variable. We examine three different cases of 10,000, 80,000, and 300,000 observations. The second example has 113 and 143 levels for the categorical variables and we compare the runtimes and speed-up for 80,000, 200,000 and 500,000 observations.

For these experiments, from a model point of view, we used the asymmetric main and interactions two-way Anova. Regarding the technical specifications, a Standard D8s v3 Azure instance with 8 cores and 32 GB memory was used.

Looking in Table 6.1 we can see that in the smallest example that consists of 15×20 levels and 10,000 observations, parallelism had the opposite results from those expected. Not only there was no speed-up, on the contrary it caused significant slow down in the execution time. This stems from the fact that in order for parallelisation to improve runtime the dataset should be big enough so that the computational cost of creating multiple tasks, running parallel threads and combining the results, would outweigh the serial simplicity. There is no golden rule for the size of data. Experiments can clarify, for each particular problem, which approach is more efficient. Indeed, we can see that as the number of observations increases, the speed-up improves significantly. Similarly, in the bigger example of 113×143 levels, the speed-up is higher for bigger datasets. Consequently, parallelism has an important effect on execution times when the volume of data is big. From the results we can see that in some cases the parallel code is three times faster than the serial. Conversely, small amounts of data constrain its performance gain and can even lead to negative effects. Concerning the difference between the two serial versions, we can confirm that in both cases the concurrent safe approach is slower and therefore its parallelisation leads to higher speed-up.

Finally, we can calculate the theoretical maximum speed-up defined by Amdah’s (Equation 6.1) and Gustafson’s (Equation 6.2) laws and see if our results are in accordance with these measures. Even though these formulae were originally developed many years ago, researchers still often use them as reference points when they want to reason about speed-up (Al-hayanni *et al.*, 2020). VisualVM is a tool for acquiring detailed information about applications that run on the JVM during their runtime. Using VisualVM for profiling and seeing the proportions of runtimes for the concurrent safe serial approach, we conclude that the proportion of runtime for the code that we decided to implement in serial is 0.57 and the proportion of code that we chose to parallelise is 0.43. Then, according to Amdahl’s law, the potential maximum speed-up is 1.6. In Table 6.1 we can see that in both examples, some of the actual speed-ups are close to that value. As the problem size increases, the speed-ups tend to increase and approximate Gustafson's theoretical speed-up estimation which after substituting our measurements in Equation 6.2, is 4.01.

6.3 Stan

Stan constitutes a probabilistic programming language that uses HMC for Bayesian inference. It is named after Stanislaw Ulam, one of the main contributors in the development of the Monte Carlo method. Its syntactical features are similar to those of JAGS, but with higher complexity. It provides an interface for R, through the 'rstan' library, but it also has an API for Python.

Stan is written in C++ and it is free and open source. A Stan program is written in a text editor or RStudio, using the syntactic rules of the language and saved in a .stan file. When running a Stan program, the parser first checks the syntactic correctness of the code, which is then translated to C++ source code and compiled to an executable file. The resulting "stanfit" object contains the output from fitting the Stan model, and can be further processed using convergence diagnostics and tools for manipulating the results.

Stan uses HMC and more specifically a no-U-turn (NUTS) sampler which is an extension of the HMC, developed by Hoffman and Gelman (2011). NUTS aims to improve the performance of the algorithm by eliminating the user-specified tuning parameters, stepsize ϵ and desired number of steps L.

Even though for a user experienced in JAGS or BUGS, writing a program in Stan is a familiar process, making Stan more efficient demands deeper understanding of the underlying algorithms and usage of programming techniques and statistical transformations that render Stan more challenging and difficult to use. Stan spends the majority of its runtime computing the gradient of the log probability function and using differentiation for these calculations is computationally demanding. Furthermore, difficult posterior geometries add extra computational cost. Two of the main suggestions in order to speed up statistical inference in Stan are to use vectorisation and reparameterisation. The former requires using vectors or arrays instead of loops when possible, to facilitate the build-in operations for gradient calculations. The latter aims to transform distributions with more complex geometries to simpler that will require a more reasonable number of steps to enter the higher probability regions. For example, approaching a heavy-tailed Cauchy using a standard uniform distribution (Betancourt, 2018*b*), or using a centred parameterisation when using a normal distribution can lead to a better effective sample size, depending on the model and structure and size of data (Stan, 2019*e*).

In this project, Stan was examined as an alternative method to implement variable selection using the Horseshoe prior, presented in Section 3.3.3. The standard Horseshoe approach, for the particular model that we examine, led to poor mixing and warnings to adapt the step-size (Stan, 2019*a*) and the depth of the trees that the algorithm evaluates during each iteration (Stan, 2019*d*). Adapting these parameters improved mixing, led to faster convergence but also to significant sampling delays especially for larger examples tried out. Keeping the parameters adapted and replacing the standard Horseshoe with the regularised Horseshoe (2.14) led to improvement in mixing and execution time. In order to decide which implementation is the most efficient for our hierarchical two-way Anova model, we finally examined the regularised Horseshoe with the default settings for the step-size and the tree-depth, which improved the runtime but deteriorated the mixing and the effective sample size. Therefore, the regularised Horseshoe with the adapted parameters appears to be the most appropriate approach for our model of interest in the particular example considered.

6.4 Rainier

In Section 5.4.2 we presented Rainier, a probabilistic programming library for Bayesian inference in Scala that uses HMC. For implementing variable selection with the Horseshoe prior we use its fastest version 0.2.2. The implementational logic follows the norms of Listing 5.4 with increased complexity deriving from the additional main effects and interaction terms.

Rainier has its own implementation for Cauchy distribution. Nevertheless, at the time of this research project, a half-Cauchy, necessary to add a Horseshoe prior on important parameters (λs and τ_{HS}) for this variable selection approach, is not yet embedded in the library. It is suggested by Rainier's author to approximate a half-Cauchy using the absolute value of a parameter that follows a Cauchy distribution. However, this can potentially affect diagnostics since some chains place parameters in positive and others in negative space, which can inflate the between-chain variance (Bryant, 2020).

6.5 Summary

In this chapter we emphasised the implementational details of TWIiS, a Gibbs sampler developed in Scala for Bayesian hierarchical modelling and in particular for fitting a two-way Anova model with interactions and random effects that constitutes our main model of study. In addition, we looked at the implementational logic behind programs written in JAGS, Rainier and Stan which are probabilistic programming languages with corresponding libraries for statistical inference in the Bayesian framework. In the following chapter these programs are applied to a simulation study using synthetic datasets and models with varying complexity levels. The main focus is placed on the results deriving from JAGS and TWIiS with further exploration of their performance and comparison of their efficiency.

Chapter 7

Simulation study

We presented in earlier chapters that for the Bayesian hierarchical two-way Anova model that we study, two possible ways to apply variable selection on the interaction effects are using indicator variables and Horseshoe priors. Both methodologies can be implemented with a Gibbs sampler. The relevant code for fitting the models of interest was developed in JAGS using R and in Scala through an implementation of a Gibbs sampler specifically for these models, which we named TWIiS (Two-way interactions in Scala). However, adaptive shrinkage priors mostly aim to enable variable selection in algorithms that support only continuous variables, such as Hamiltonian Monte Carlo. Therefore, two additional implementations were also considered, using Stan in R and Rainier in Scala, both based on HMC.

This chapter is dedicated to exploring, through a simulation study, examples with varying sizes and complexity for both variable selection approaches. Out of all implementations examined, the main focus is placed on TWIiS and JAGS. For these programs we are not only interested in assessing their actual results but also in evaluating their efficiency and comparing their performance. This comparison is valuable to decide which methodology appears to perform better for the type of modelling that we examine.

7.1 Design of the simulation process

In order to ensure that TWIiS is implemented correctly and to validate its results against JAGS, synthetic datasets were simulated and used for modelling in both cases. Variable selection for the interactions of the two-way Anova models, as presented in Chapter 3, is applied for each of the four cases: “asymmetric main and interaction effects”, “symmetric main and asymmetric interaction effects”, “ asymmetric main and symmetric interaction effects” and “symmetric main and interaction effects”. Therefore, synthetic datasets must be simulated for each case. In all cases the models involve two independent categorical variables and their interactions.

The aim was to simulate three different examples for each case, with varying sizes of levels for each categorical variable, to assess the results and performance of programs as model complexity increases. The sample size was set to 10,000 observations. The first example has 15 levels for the first categorical variable denoted as α and 20 levels for the second denoted as β, for the cases that include asymmetric main effects. The number of potential interactions is $15 \times 20 = 300$. The example is referred to as 15×20. When the main effects are treated as symmetric, we assume that there is only one categorical variable denoted as z. Since the model is a two-way Anova, z is present twice, with its level being either the same or different for the first and the second categorical variables for each observation. The number of levels for z for each example is set to be the maximum value between the numbers of levels for α and β. Hence, in the example of 15×20, for the cases that treat the main effects as symmetric, z has 20 levels and is referred to as 20×20. The other examples simulated are 50×60 and 113×143. For the latter we also simulated a dataset with 80,000 observations in order to approach the number of levels and observations that the actual dataset of the yeast genome case study has.

For all examples we set the levels 9 and 14 for variable α to be "missing". Hence, there were no observations in the simulated datasets that have either value. This was a deliberate choice in order to assess the behaviour of the code and ensure that it works properly for "missing" data, since the real dataset has some combinations missing.

The synthetic data simulation process was implemented in R. It is similar for all cases and consists of the following steps:

1. Definition of sample size, and number of levels for the first (n_j) and the second categorical variables (n_k). Usage of flags to denote whether the main or interaction effects are treated as symmetric.

2. Sampling with replacement the "observed" levels for each variable per observation. Levels 9 and 14 for the first categorical variable are excluded.

3. The error for each observation is sampled from a standard normal distribution N(0,1) and the intercept term is fixed to 3.1.

4. The variances of the main effects, σ^2_α and σ^2_β, are sampled from a Gamma distribution $Gamma(4, 1)$ and used to sample the actual main effects for each of their levels from a normal distribution, $N(1.5, \sigma_\alpha)$ and $N(5.0, \sigma_\beta)$ respectively.

5. Using an inclusion probability $p = 0.2$ for the cases where the interaction effects are asymmetric we fill a matrix of size $n_j \times n_k$, which is the number of potential interactions, with samples from a binomial distribution that show which interactions should be included in the model.

For the cases that treat the interaction effects as symmetric the inclusion probability is $p = 0.1$ and only the lower diagonal matrix is filled with binomial samples, with the symmetric values being set correspondingly equal.

6. The variance of the interaction effects is sampled from $Gamma(4, 1)$, and further used to sample the interaction effects from a normal distribution, $N(1.5, \sigma_\gamma)$. The interactions for the coefficients that should not be included in the model are fixed to zero.

7. The response for each observation is calculated after adding the intercept value, the main effects for the first and the second categorical variables, the corresponding interaction effect and the error.

8. The synthetic dataset is then exported to a CSV file. The expected values are also saved and used in the posterior analysis to evaluate the results.

Finally, it is worth adding that as the number of levels of the categorical variables increases in bigger examples, some combinations of levels do not have any observations. For these cases and for the cases where we deliberately excluded some levels (i.e. levels 9 and 14 for α) the corresponding interactions will be highlighted in the relevant plots presented in the following sections.

7.2 Posterior analysis methods

In the examples of the following sections, the results are summarised through various tables. For validation and comparison of the sampling outcome we are using convergence diagnostics and plots, colour-coded heatmaps, and MCMC efficiency measurements. Following some convergence diagnostics and metrics mentioned in 2.2.4, the methods used in this chapter are detailed below:

- Convergence diagnostics and plots

  To ensure that after a long run the MCMC has converged we look at some diagnostics plots such as the traceplots and aurocorrelation plots, and the *Effective Sample Size (ESS).*

  Regarding the traceplots, no visible anomalies are expected to be observed. Good mixing indicates that the Gibbs sampler has explored adequately the sample space multiple times. The autocorrelation function (ACF) plot must have a declining tendency indicating that there is a decreasing sample autocorrelation among the draws of the variable examined as the lag increases. Consequently, the further apart two samples are in the chain the more independent they are, which is an important and desirable characteristic to conclude that the results are sufficiently independent and represent quite accurately the target posterior distribution.

Ideally, the random samples drawn from complex distributions should be independent. Even though this is not feasible for the MCMC, there is an estimate to assess a notion of "ratio" between the dependent samples drawn and the amount of the equivalent independent ones, which is called *Effective Sample Size (ESS)*. ESS is based on the number of samples (n) and the autocorrelation at lag k ($\rho(k)$). It is defined as:

$$ESS = \frac{n}{1 + 2\sum_{n=1}^{\infty} \rho(k)}.$$

The ESS is actually an estimate of the sample size needed to attain an equal level of posterior accuracy if that sample was a simple random independent sample. Usually the ESS increases as the number of iterations increases. The closer the ESS to the actual number of samples the weaker the autocorrelation between the samples. Ideally the ESS for efficiency comparisons should be estimated based on the posterior samples drawn without thinning after the algorithm has achieved convergence. Thinning reduces autocorrelation in the samples and can produce a higher effective sample size than the unthinned alternative (Stan, 2019*b*). Due to the large amount of the MCMC output that can lead to increased memory requirements and limit the storage and processing capacities, thinning is sometimes necessary. Thinned samples that retain autocorrelation can also be used to reach a consensus about this measure. The minimum ESS across all variables in the model is considered in order to compare the parameter which is the least efficient.

- Efficiency measurement

  The efficiency of MCMC algorithms depends not only on the computational speed but also the mixing. A metric which constitutes a combination of these two notions is the *ESS/sec*, thus

$$\text{MCMC efficiency} = \frac{\text{effective sample size}}{\text{computation time (in sec)}}.$$

  Hence, it is the number of independent samples per second. This metric is useful to evaluate the performance of different MCMC implementations, and we will use it to compare the alternative programs explored.

- Test of equality between the posterior densities

  To assess whether the inferred posterior distributions from two or more implementations are similar, we can visualise and observe their densities after overlaying them. The plots will then display the moments and the tail behaviour, and if they overlap they confirm that the two inferred posterior densities are alike. In terms of exploring the equality of posterior distributions of various programs that we examine, in this thesis we adopt an approach where we examine and overlay the univariate marginal posterior distributions for numerous variables. Even though it is technically possible that the univariate marginal posterior distributions of two implementations could be similar but the full posterior distribution could be different, this is very unlikely. However, we have to note that equality of the univariate marginal posterior distributions does not absolutely guarantee equality of the full posterior distributions, but strongly suggests their similarity. Therefore, we compare numerous univariate marginal posterior distributions produced from our programs of interest by overlaying them and provide examples of these comparisons throughout the relevant sections of this chapter.

In the literature, there are also more formal hypothesis tests performed to verify the similarity between the two densities, including the Kolmogorov-Smirnov test and the permutation test of equality. Supposing that the posterior distribution estimated from one program is denoted as $F_1(x)$ and from a second as $F_2(x)$, the null hypothesis for the two tests is that the two distributions are the same. For the Kolmogorov-Smirnov test, which is available in the R package "stats", the null hypothesis is accepted or rejected based on the statistic $q = max_x|F_1(x) - F_2(x)|$. The second test, is also available in R through the library "sm", which calculates the test statistic as the sum of the squared differences in the density estimates $ts = sum((F_1 - F_2)^2)$. The permutation test of equality involves calculation of the initial test statistic and numerous permutations at the posterior estimates of two programs through sampling. For every permutation considered the corresponding test statistic is calculated and compared to the initial. The p-value is then calculated as the proportion of test statistics that were greater than the initial.

7.3 Variable selection using indicators

In the following sections, we present and compare the results and performance of the Gibbs sampler for a two-way Anova implemented in TWIiS and JAGS through three examples using synthetic datasets. Variable selection is applied on the interactions using variable selection indicators and more precisely the Kuo and Mallick approach. The first example involves a model with a small number of levels for the categorical variables in order to graphically illustrate the results in a clearer way. The second and third examples expand the comparison to bigger models to demonstrate how the results and runtimes of the two blocks of code evolve for more complex and computationally demanding problems. Regarding the computing resources, a Standard D8s v3 Azure instance with 8 cores and

32 GB memory was used.

7.3.1 Example 1: Synthetic dataset with 15×20 levels

In this example we examine a two-way Anova model with interactions where in the cases of asymmetric main effects the first categorical variable α consists of 15 levels and the second variable β of 20. For the cases where the main effects are symmetric, both variables are denoted with z and have 20 levels. The process followed to simulate this dataset was discussed in Section 7.1. The methodology used to apply variable selection on the interactions in every case considered is the Kuo and Mallick approach. The low number of levels facilitates the graphical representation of the estimated interaction terms through various heatmaps that allow the display of which interactions are considered important after applying variable selection, the strength of these interactions and the estimated group means.

Initially, running the algorithm for 100,000 iterations with a thinning factor of 10 resulted in good convergence diagnostics for the interaction coefficients and precisions, but showed inadequate mixing and high autocorrelations between the samples for the main effects and μ for both JAGS and TWIiS. Increasing the number of iterations to one million and adapting the thinning factor to a hundred led to convergence and improved the mixing of the main effects and the mean. Regarding the equivalent autocorrelation values, they remained high however decreasing for both programs. The ESS/sec used as a measurement of efficiency comparisons between JAGS and TWIiS is calculated and derived from this run. At ten million iterations with respective adaptation of the thinning factor the mixing and the within-chain autocorrelations were significantly improved for the overall mean and the main effects. Nevertheless, the effective sample size was relatively low for μ and some main effects, such as the second categorical variable β in the case of asymmetric and z in the case of symmetric main effects. Further increase in the number of iterations of the Gibbs sampler to 100 million has a double purpose. Firstly, this long run leads to additional reduction of the earlier autocorrelation between the samples. Secondly, it can lead to safe conclusions about the proximity of the posterior densities estimated from both programs and subsequent verification of the correctness of the Gibbs sampler developed in Scala. The inference results as well as the graphical depiction presented in the following sections are the outcome of this long MCMC run, which requires longer runtimes but also leads to significant improvement in earlier inefficiencies. Burn-in period and thinning that led to 9,900 MCMC samples were applied to all cases.

Of the four different cases of asymmetry considered, first we examine thoroughly the case with both the main and interaction effects treated as asymmetric. The analysis of the results for the remaining cases is less detailed but aims to highlight the key points of comparison between JAGS and TWIiS.

7.3.1.1 Asymmetric main and asymmetric interaction effects

In the case where both the main and interaction effects are asymmetric we assume that each of the two independent categorical variables comes from a different distribution and the effect of the interaction γ_{jk}, defined as the result of the product of the variable selection indicator I_{jk} and the strength of the interaction estimate θ_{jk}, is different from the effect of the interaction γ_{kj}. Recalling the model for this case from 3.3.2.1 it is specified as:

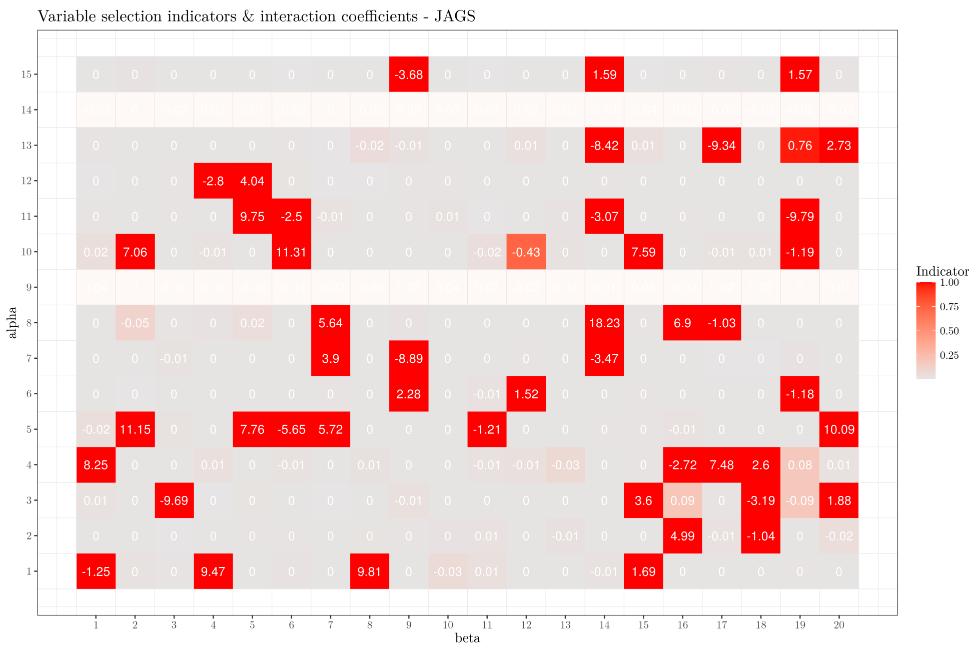
Model

$X_{ijk}|\mu, \alpha_j, \beta_k, I_{jk}, \theta_{jk}, \tau \sim \mathrm{N}(\mu + \alpha_j + \beta_k + I_{jk}\theta_{jk},\ \tau^{-1})$, i=1,...,$N$,

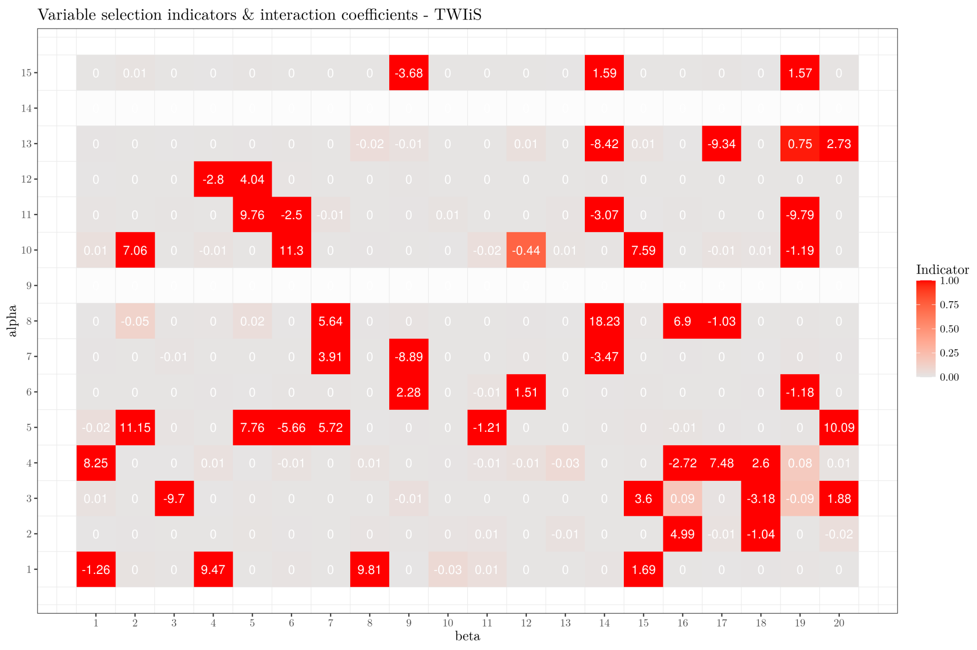
where N is the number of observations.

Results

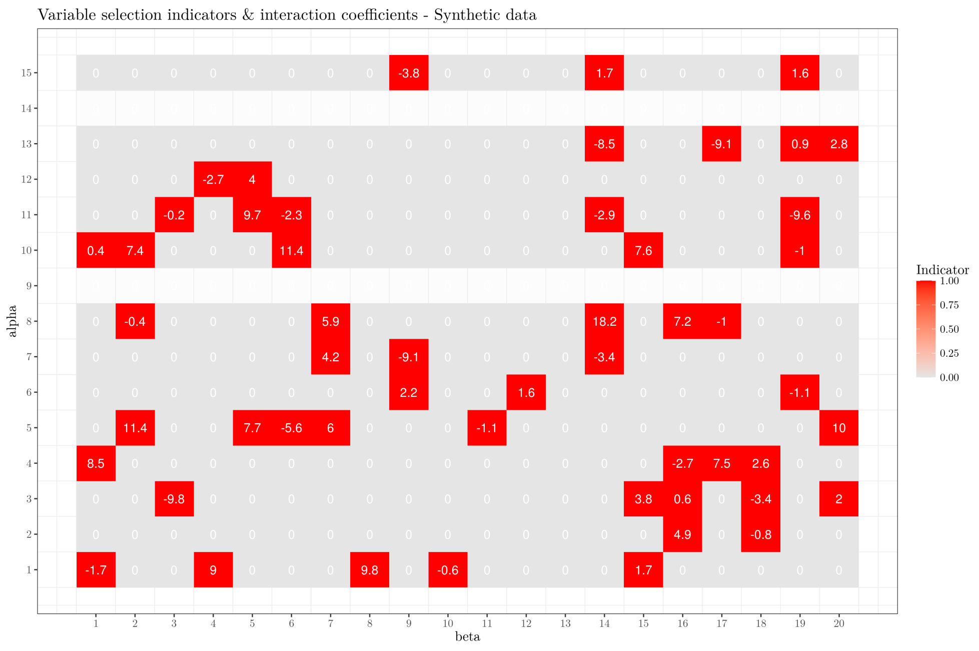
The returned results for the interaction coefficients estimated are presented graphically in Figure 7.1. We can observe that both TWIiS and JAGS highlight the same interactions as important. The strength of the indicators is colour-coded, with cells in grey representing the interactions with inclusion probability close to zero and therefore considered as less significant. On the other hand, cells in red indicate high inclusion probabilities and point out the interactions that should be included in the model. Comparison between the heatmaps of the two implementations and the heatmap representing the true interaction effects (7.1c) confirms that they are very similar. In all heatmaps we can see that the interaction effects for the levels 9 and 14 of the first categorical variable α are marked with white. This indicates that the simulated dataset did not contain any observations with these levels for the first categorical variable. “Missing” levels are imported deliberately to ensure that the algorithms will work properly for datasets potentially having this characteristic. The two methodologies handle the “missing” levels differently during MCMC sampling. In TWIiS the Gibbs sampler is implemented to assign zero to the effects involving “missing” levels, whereas JAGS samples from the prior distribution of these variables. This implementational difference is also visible in Figure 7.2 that illustrates the mean and standard deviation of each main effect estimate. In this plot we can also see that there is similarity in the other estimates produced by the two programs. Comparison of the results to the true main effects reveals that there is a divergence between the estimated values from both TWIiS and JAGS and the equivalent expected true values as shown in Table 7.1 for some indicative and randomly chosen variables. Most probably, this an identifiability problem due to the small number of levels in this example. Nevertheless, in the same table we notice that the resulting group means estimated from TWIiS and JAGS are in close proximity to the true equivalent values. Graphical representation of the group means estimated from both programs compared to the true group means is available in Figure 7.3, where it is shown that the three heatmaps follow the same pattern.



(a) JAGS



(b) TWIiS

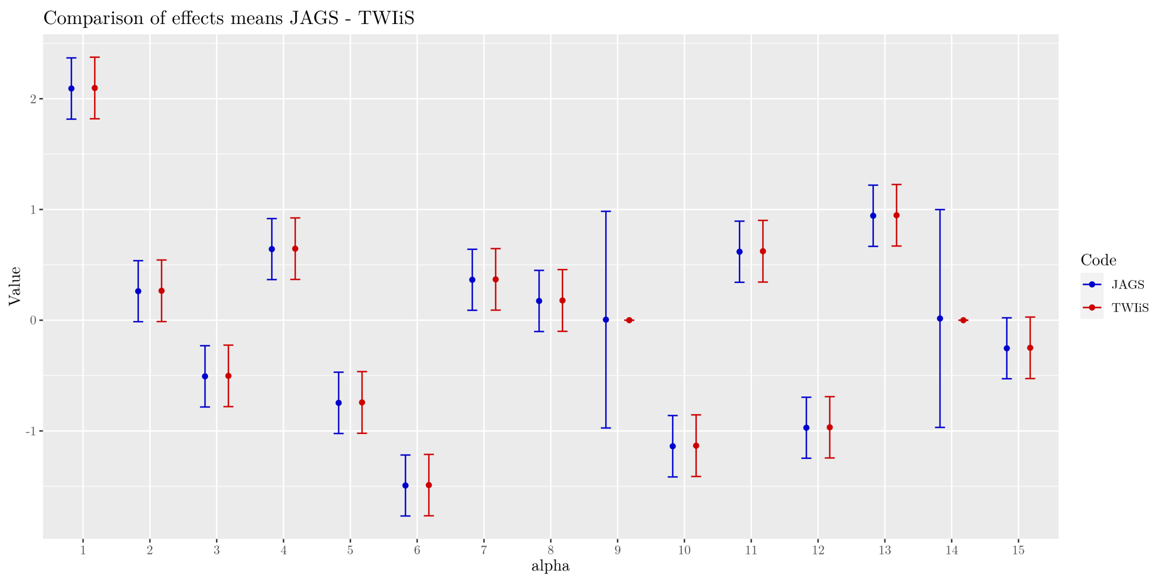


(c) True coefficients

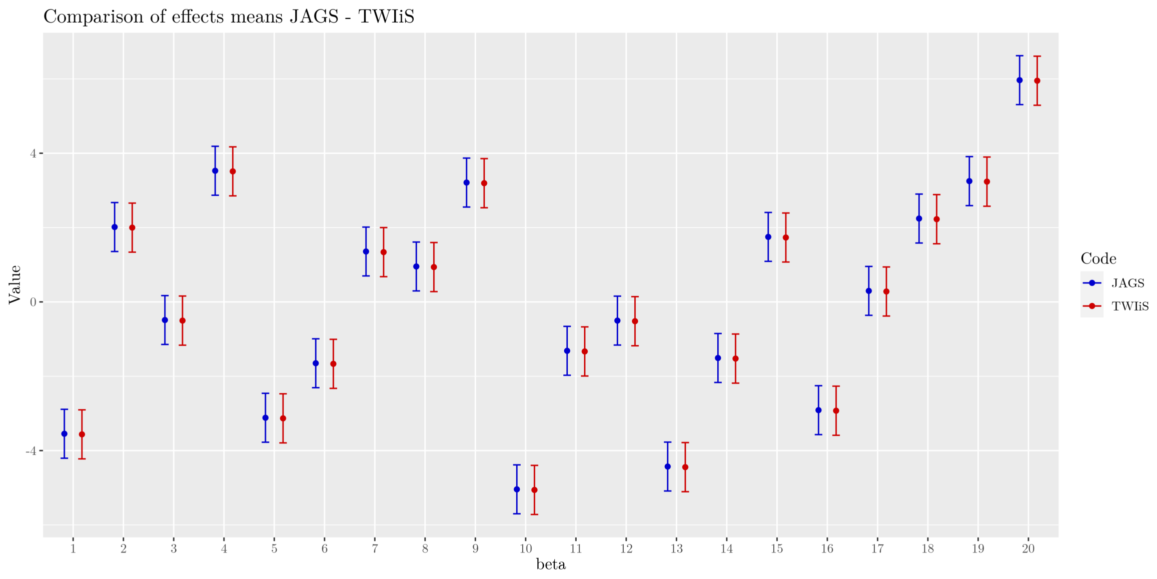
Figure 7.1: Comparison of the results among JAGS, TWIiS and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable.
Case: Asymmetric main & interaction effects

Table 7.1: Comparison of some effects and group means among JAGS, TWIiS and the true simulated coefficients.
Case: Asymmetric main & interaction effects

Variable	JAGS	TWIiS	Simulated
α_1	2.100	2.096	4.0
α_{10}	-1.133	-1.134	0.6
β_4	3.523	3.543	8.4
β_{15}	1.742	1.751	6.5
$\gamma_{1,4}$	9.467	9.467	9.0
$\gamma_{10,15}$	7.591	7.588	7.6
μ	9.704	9.699	3.1
τ_a	1.193	1.201	2.0
τ_b	0.125	0.125	5.3
$\mu + \alpha_1 + \beta_4 + \gamma_{1,4}$	24.794	24.805	24.5
$\mu + \alpha_{10} + \beta_{15} + \gamma_{10,15}$	17.904	17.904	17.8



(a) Comparison for αs

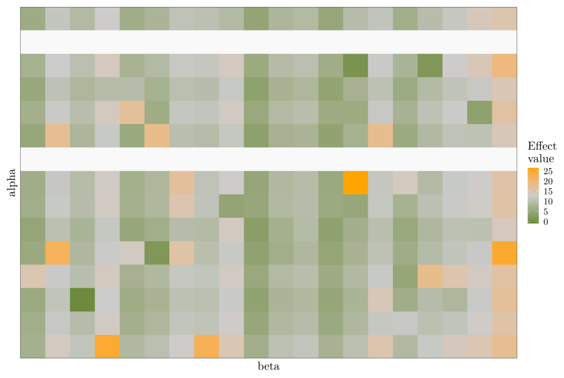


(b) Comparison for βs

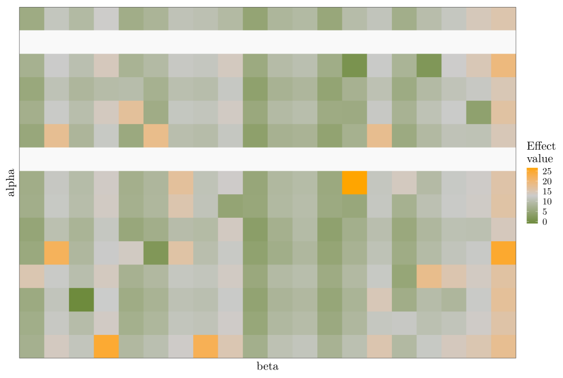
Figure 7.2: Comparison of main effects means estimated from JAGS and TWIiS.
Case: Asymmetric main & interaction effects

Convergence diagnostics and plots

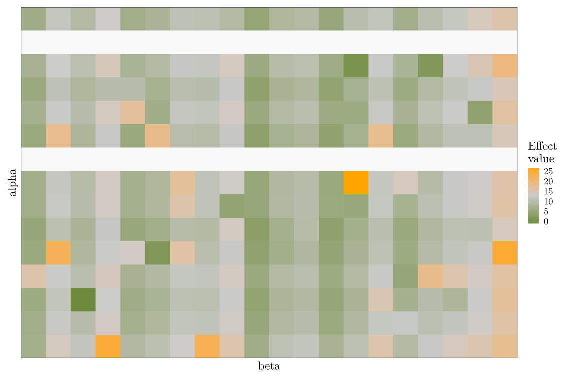
To verify that the MCMC has achieved convergence after a long run we look at some diagnostic plots such as the traceplots of some randomly selected estimated variables presented in Figure 7.4, where we can see that there are not any visible anomalies. There is a good mixing and the Gibbs sampler appears to have explored adequately the sample space for all the parameters in both programs. The autocorrelation function (ACF) plots



(a) JAGS



(b) TWIiS



(c) True coefficients

Figure 7.3: Comparison of the group means among JAGS, TWIiS and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable. *Case: Asymmetric main & interaction effects*

displayed in Figure 7.5 indicate that there is a decreasing sample autocorrelation of the draws for all the covariates examined as the lag increases for both JAGS and TWIiS. Consequently, there is a decaying dependence between the samples which indicates that they represent with sufficient accuracy the stationary distribution.

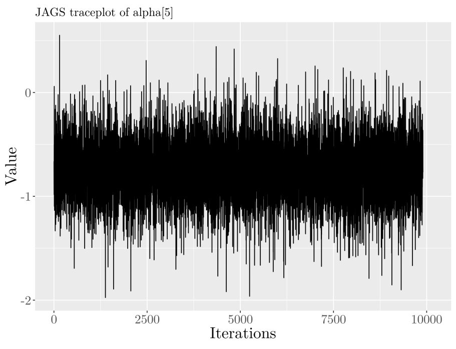
Another heuristic of the posterior analysis of the MCMC is the effective sample size. It accounts for the autocorrelation and dependence between the samples produced during the MCMC run and denotes the number of corresponding equivalent independent samples. Table 7.2 summarises the ESS estimated at different number of iterations of the Gibbs sampler for the variables that presented the minimum ESS in their group. To be more specific about which variable is identified as having the minimum ESS we also include their index. For example at one million iterations for the first categorical variable α, α_{12} has the smallest ESS for JAGS, whereas α_8 is the variable with the smallest ESS for TWIiS. However, as long as we compare the smallest ESS for each category of variables, the index is of no importance.

For all cases, the first 7,000 samples after the algorithm achieved convergence were used to produce the efficiency metrics. We can observe that for the variables that have low ESS for the runs with one million iterations, such as the αs, the ESS increases as the number of iterations and thinning increase. This is caused by the fact that long runs and thinning reduce autocorrelation and produce higher effective sample sizes. We consider as a reliable ESS estimate for efficiency comparisons, a value that derives from a chain that has converged and is either unthinned or thinned but still retaining autocorrelation, such as in the case of one million iterations. Furthermore, it is important to look at the minimum ESS across all variables in the model. As an example we can consider the minimum ESS for one million iterations, which is 69.29 for μ and derives from JAGS. An ESS value of 69.29 suggests that the 7,000 samples of the Markov chain correspond to 69.29 independent samples. For inference, the ESS must be close to the sample size.

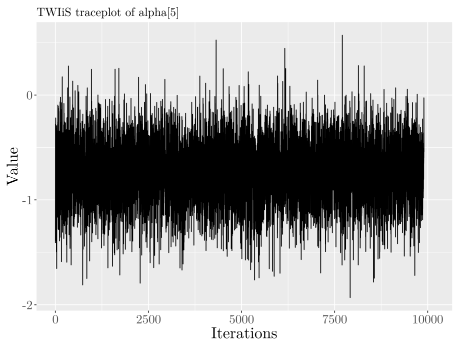
One other aspect that was taken into account during the implementation of variable selection using the Kuo and Mallick approach is the decision upon the prior assigned to the inclusion probability p. Knowing that the model is sparse and therefore most of the prior mass should be less than 0.5, a $Beta(2, 10)$ was chosen as a fairly weak but informative prior. This prior encourages the model to explore a sensible range of ps and allows it to determine from the data the inclusion probability that best describes the sparsity. The comparison between the prior distribution of p and its estimated posterior distribution from TWIiS and JAGS can be seen in Figure 7.6. Knowing that the true value of the inclusion probability used to simulate the synthetic dataset is 0.2, we can see that the posterior mass is focused at that point and therefore the model has successfully estimated the inclusion probability that best describes the sparsity of interactions.

Test of equality between the posterior densities

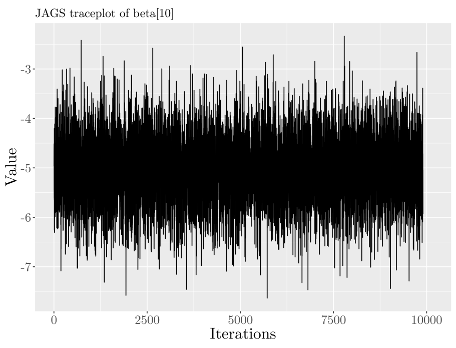
To determine whether the inferred posterior distributions deriving from the two programs are similar we can visualise and observe the two densities after overlaying them to compare their moments and tail behaviour. As mentioned in 7.2 similarity in the univariate marginal posterior distributions does not absolutely guarantee similarity in the full posterior distributions. In the scope we are examining, we plot, overlay and compare numerous univariate marginal posterior densities to assume agreement of the full posterior distributions. Examples of these plots for some of our variables are shown in Figure 7.7, and since they overlap we consider them alike. Further confirmation of the visual evaluation of convergence was achieved, when necessary, through the Kolmogorov-Smirnov test and the permutation test of equality. Consequently, we can conclude that in the case of the two-way Anova with application of variable selection on the interactions and the assumption that both the main and the interaction effects are asymmetric, JAGS and TWIiS produce similar posterior distributions of draws.



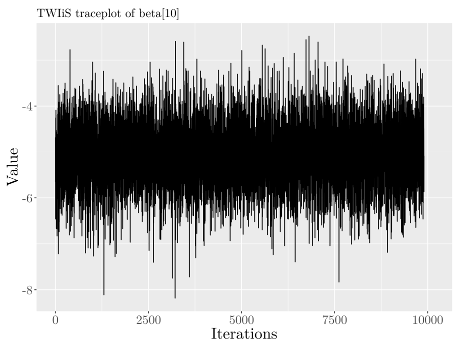
(a) JAGS - effect α_5



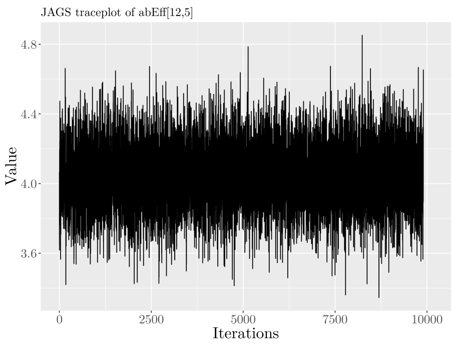
(b) TWIiS - effect α_5



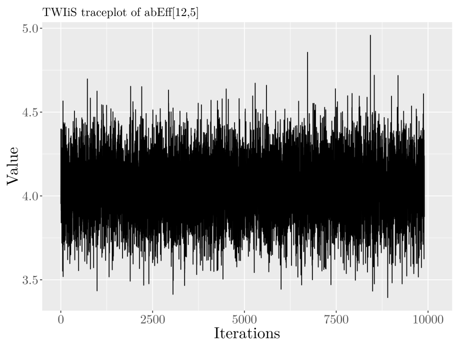
(c) JAGS - effect β_{10}



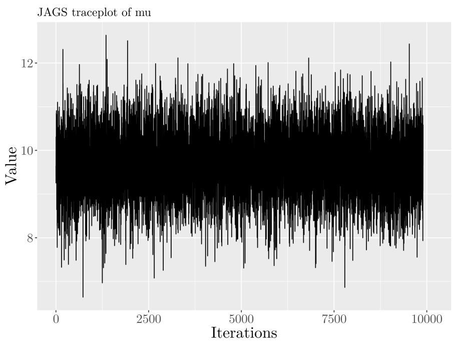
(d) TWIiS - effect β_{10}



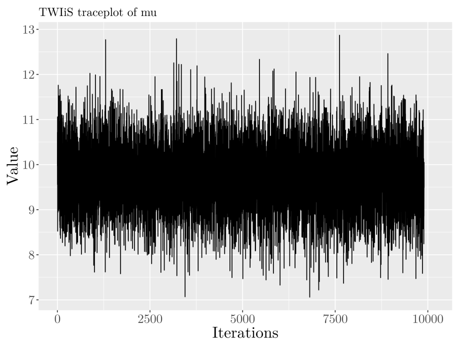
(e) JAGS - interaction effect $\gamma_{12,5}$



(f) TWIiS - interaction effect $\gamma_{12,5}$

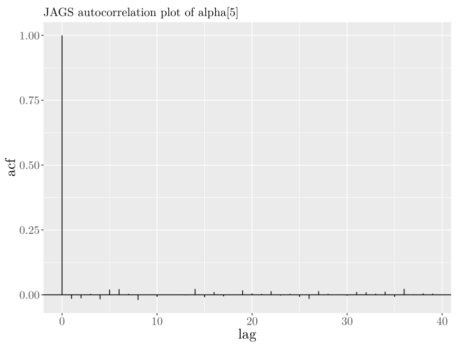


(g) JAGS - μ

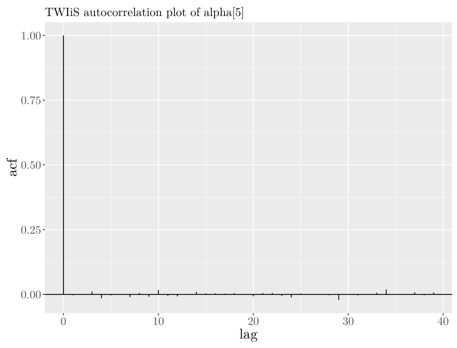


(h) TWIiS - μ

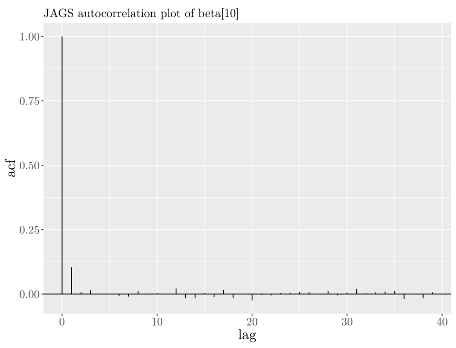
Figure 7.4: Traceplots from JAGS and TWIiS. The y-axis represents the value and the x-axis the iterations.
Case: Asymmetric main & interaction effects



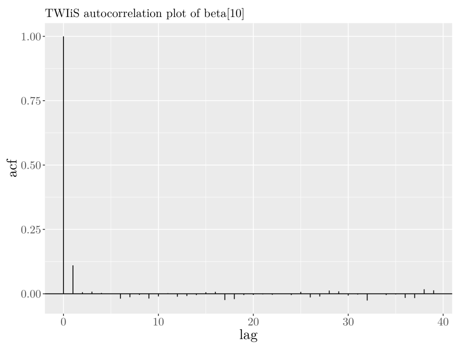
(a) JAGS - effect α_5



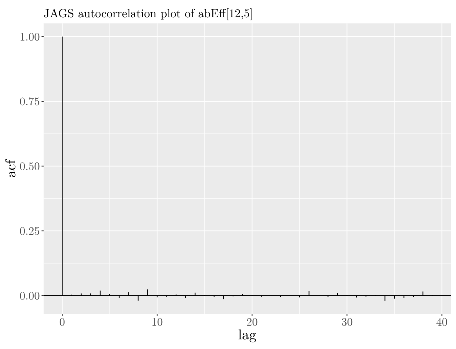
(b) TWIiS - effect α_5



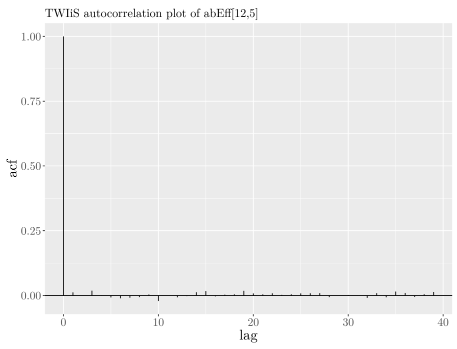
(c) JAGS - effect β_{10}



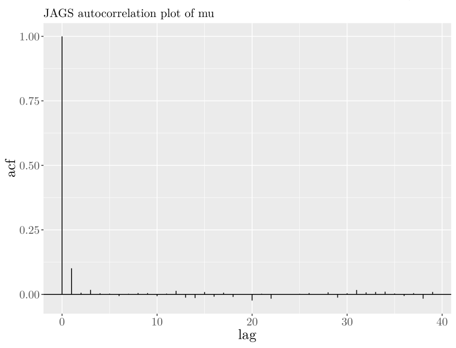
(d) TWIiS - effect β_{10}



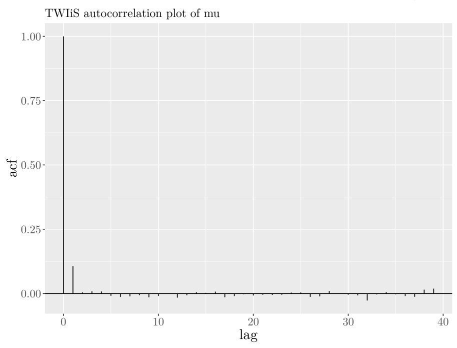
(e) JAGS - interaction effect $\gamma_{12,5}$



(f) TWIiS - interaction effect $\gamma_{12,5}$



(g) JAGS - μ



(h) TWIiS - μ

Figure 7.5: Autocorrelation plots from JAGS and TWIiS. The y-axis represents the autocorrelation value and the x-axis the lag.
Case: Asymmetric main & interaction effects

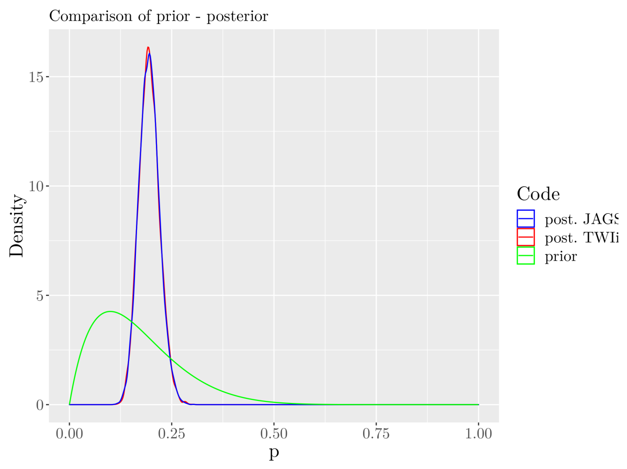
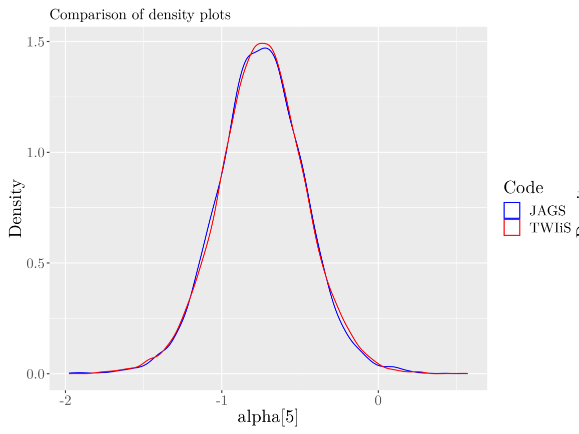
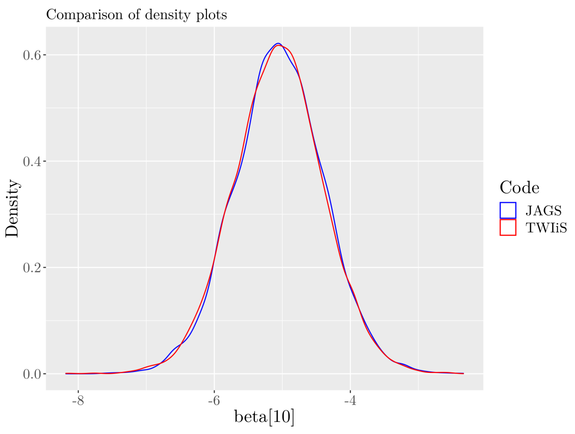


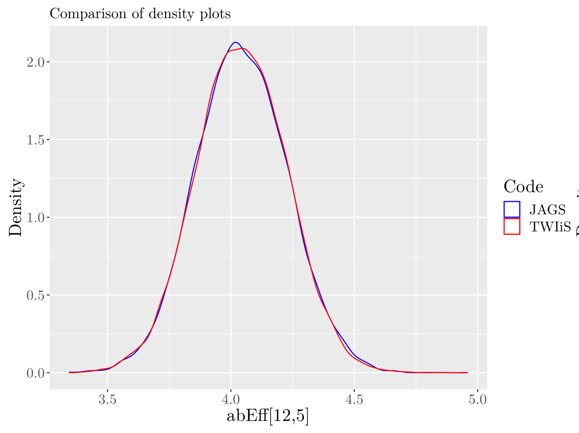
Figure 7.6: Comparison of the prior and the posterior distributions for p.
Case: Asymmetric main & interaction effects



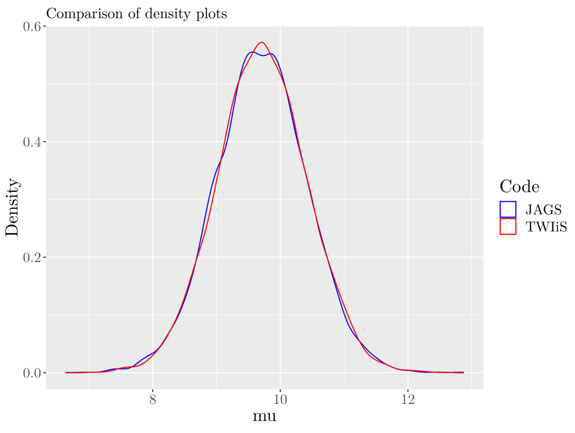
(a) Overlaid densities - effect α_5



(b) Overlaid densities - effect β_{10}



(c) Overlaid densities - interaction effect $\gamma_{12,5}$



(d) Overlaid densities - μ

Figure 7.7: Posterior density plots from JAGS and TWIiS.
Case: Asymmetric main & interaction effects

Efficiency measurement

In order to assess the efficiency of MCMC algorithms ESS/sec is used as a metric that accounts for both the actual runtime of the algorithm and the level of mixing. ESS/sec represents the number of independent samples per second. This metric is useful to evaluate and compare the performance between different MCMC implementations such as JAGS and TWIiS in our case. The runtimes used for producing the results deriving from TWIiS in this and the following sections correspond to the actual runtime of the serial version of the program as described in Section 6.2.4. The last column of Table 7.2 displays the smallest ESS/sec measurements for each category of variables. As with the evaluation of the ESS that requires the minimum ESS across all variables in the model to be accounted for, the minimum ESS/sec has to be identified and compared. We can see that, in this particular example with 15$\times$20 levels, the Gibbs sampler implemented in Scala at one million iterations presents the smallest ESS/sec for the variable μ that has the value 0.286. For JAGS the minumum ESS/sec is for the same variable evaluated at 0.030. This difference indicates that TWIiS performs better than JAGS by generating more effectively independent samples per second. Furthermore, their ratio is 9.53 which suggests that TWIiS is approximately nine times more efficient than JAGS in this case. We notice that even though the effective sample sizes do not present substantial differences for the two programs, the runtime of TWIiS is significantly faster than JAGS. This fact is reflected in the ESS/sec rendering TWIiS notably more efficient for the model with the certain characteristics assumed in terms of complexity, sample size and symmetry in the effects. In the following sections we are going to explore the results, metrics and features of the remaining cases of asymmetry in order to verify that the results derived from TWIiS and JAGS are similar for all models examined and assess the efficiency of the two implementations in all cases.

Table 7.2: Efficiency measurement and comparison between JAGS and TWIiS for Example 1: 15×20. *Case: Asymmetric main & interaction effects*

	Iterations	Time (secs)	Variable	ESS	ESS/sec
JAGS	1m	2,193	α_{12}	484.22	0.221
			β_6	69.20	0.032
			$\gamma_{12,11}$	4,839.82	2.207
			μ	65.63	0.030
TWIiS	1m	242	α_8	469.33	1.939
			β_{12}	79.25	0.327
			$\gamma_{11,18}$	7,807.00	22.563
			μ	69.29	0.286
JAGS	10m	20,437	α_4	3,982.39	0.195
			β_8	706.55	0.035
			$\gamma_{12,1}$	4,673.21	0.229
			μ	789.18	0.039
TWIiS	10m	1,916	α_7	4,271.04	2.229
			β_3	748.73	0.390
			$\gamma_{6,3}$	4,742.43	2.475
			μ	739.97	0.386
JAGS	100m	200,390	α_9	6,545.75	0.033
			β_{11}	5,757.84	0.029
			$\gamma_{10,10}$	4,993.72	0.025
			μ	5,794.72	0.029
TWIiS	100m	24,682	α_{10}	7,000.00	0.284
			β_{11}	5,669.07	0.230
			$\gamma_{10,3}$	5,375.16	0.218
			μ	7,990.36	0.226

7.3.1.2 Symmetric main and asymmetric interaction effects

In the case where the main effects are symmetric and the interaction effects are asymmetric, we assume that $\alpha_i = \beta_i$ for all i and the effect of the interaction γ_{jk} is different from the effect of the interaction γ_{kj}. Both main effects are denoted as z, with z_i indicating each effect and $i = 1, ..., n_z$. The corresponding model was presented in Section 3.3.2.2 and is expressed as:

Model

$X_{ijk}|\mu, z_j, z_k, I_{jk}, \theta_{jk}, \tau \sim$ N($\mu + z_j + z_k + I_{jk}\theta_{jk}$, τ^{-1}), i=1,...,N,

where N is the number of observations.

Results

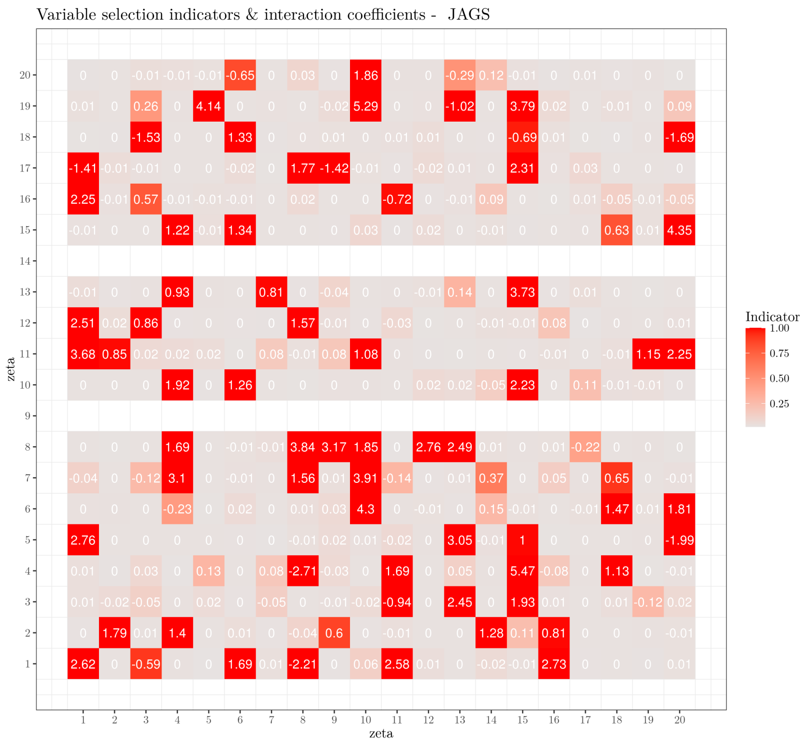
Regarding the interaction effects, comparing the heatmaps that present the estimates from

JAGS and TWIiS illustrated in Figure 7.8 shows that they follow a similar pattern and the same interactions are estimated as important to be included in the model from both applications. In addition, the values estimated from both programs are similar and at the same time close to the true interactions of the synthetic dataset.

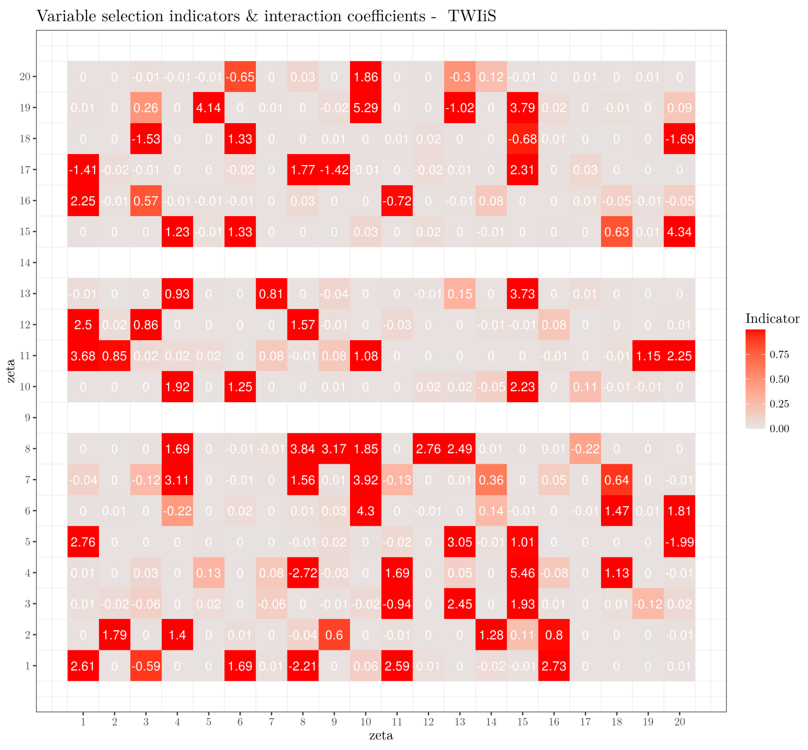
Regarding the main effects, in Table 7.3 we can see at the examples that even though the estimated coefficients from TWIiS are close to the ones estimated from JAGS, they are both different from the expected coefficients. This is possibly again due to an identifiability problem, since the group means from both programs are close to the expected group means of the synthetic dataset.

Indicative traceplots and autocorrelation plots are displayed in Figure 7.9. For both TWIiS and JAGS there is good mixing and decreasing autocorrelation. The overlaid densities presented in Figure 7.10 show that the posterior densities produced by both programs are very much alike.

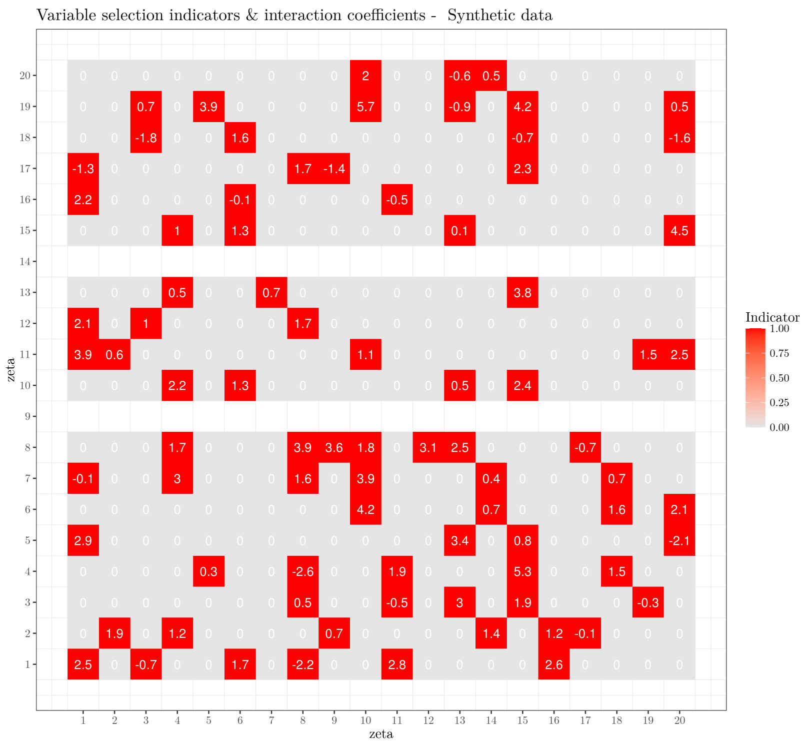
As in the asymmetric main and interaction effects case, the performance of the two programs is compared through the efficiency metric ESS/sec, which combines the runtime and the ESS and expresses the effect of sample dependence in terms of performance. Table 7.4 shows that this metric presents higher values for TWIiS, suggesting that it has better performance than JAGS for this particular example. Comparing the ESS/sec leads us to the conclusion that the TWIiS code is approximately ten times faster than JAGS.



(a) JAGS

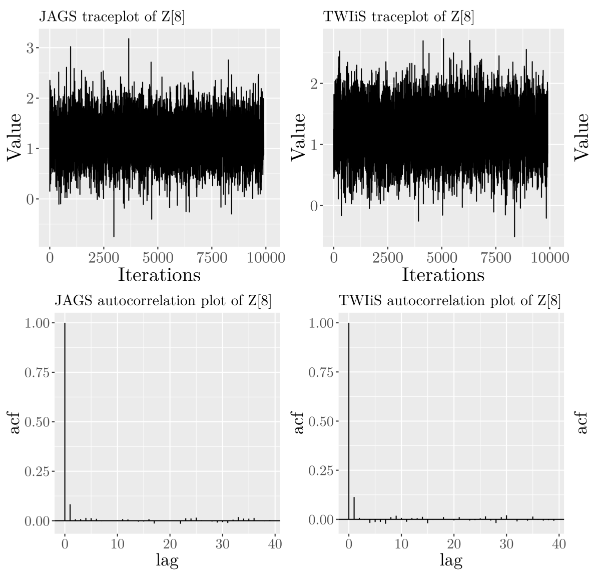


(b) TWIiS

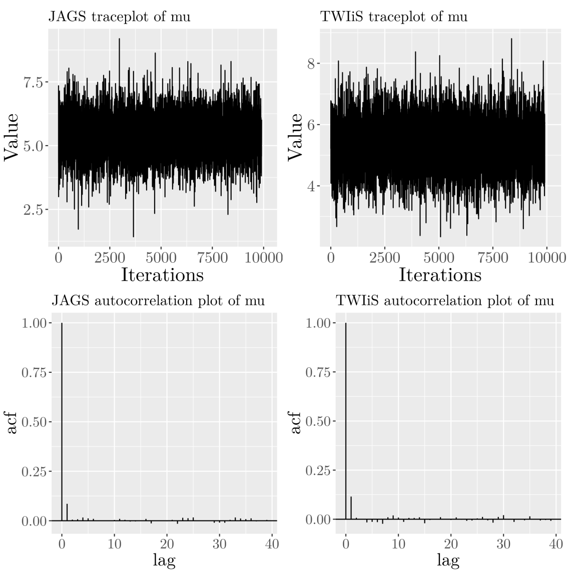


(c) True coefficients

Figure 7.8: Comparison of the results among JAGS, TWIiS, and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable.
Case: Symmetric main & asymmetric interaction effects



(a) Comparison - effect Z_8

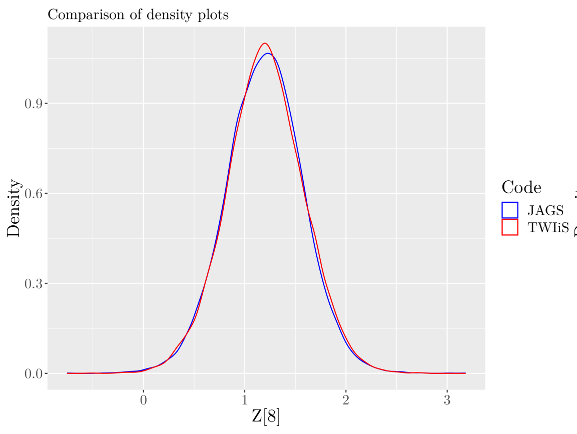


(b) Comparison - μ

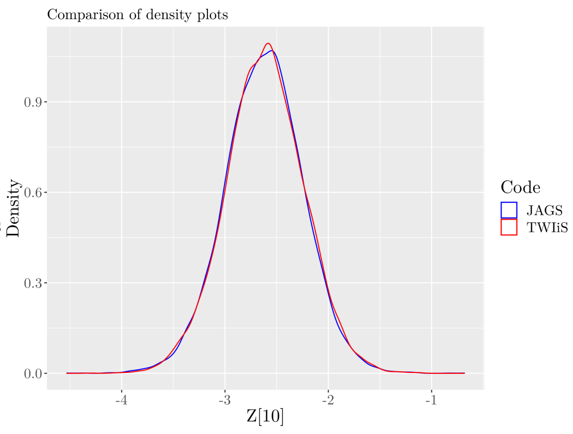
Figure 7.9: Comparison of traceplots and autocorrelation plots between JAGS and TWIiS.
Case: Symmetric main & asymmetric interaction effects

Table 7.3: Comparison of some effects and group means among JAGS, TWIiS and the true simulated coefficients.
Case: Symmetric main & asymmetric interaction effects

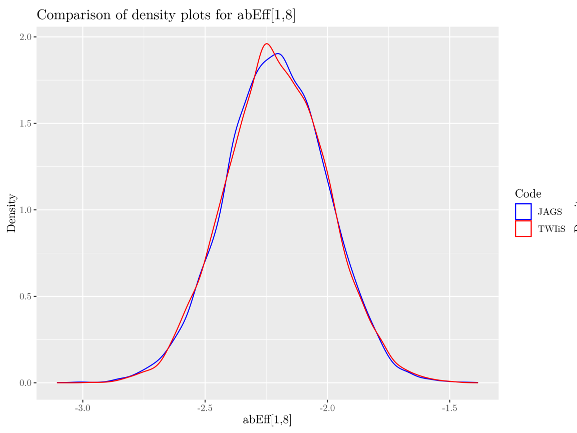
	JAGS	TWIiS	Simulated
z_1	0.157	0.163	1.3
z_5	2.140	2.147	3.3
z_8	1.195	1.203	2.3
z_9	-1.114	-1.108	0.0
$\gamma_{5,1}$	2.763	2.760	2.9
$\gamma_{8,9}$	3.173	3.170	3.6
μ	5.370	5.354	3.1
τ_z	0.388	0.387	3.5
$\mu + z_5 + z_1 + \gamma_{5,1}$	10.430	10.424	10.6
$\mu + z_8 + z_9 + \gamma_{8,9}$	8.624	8.619	9.0



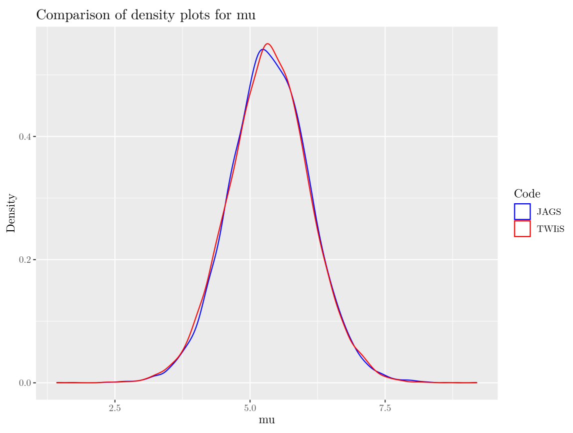
(a) Overlaid densities - effect Z_8



(b) Overlaid densities - effect Z_{10}



(c) Overlaid densities - interaction effect $\gamma_{1,8}$



(d) Overlaid densities - μ

Figure 7.10: Posterior density plots from JAGS and TWIiS.
Case: Symmetric main & asymmetric interaction effects

Table 7.4: Efficiency measurement and runtime comparison for Example 1: 15×20.
Case: Symmetric main & asymmetric interaction effects

Code	Iterations	Time (secs)	Variable	ESS	ESS/sec
JAGS	1m	2,018	Z_5	85.14	0.042
			$\theta_{20,1}$	5,890.09	2.918
			μ	85.71	0.042
TWIiS	1m	220	Z_{15}	91.72	0.417
			$\theta_{20,12}$	5,671.52	25.780
			μ	94.56	0.430
JAGS	10m	19,714.8	Z_{10}	783.88	0.039
			$\theta_{20,1}$	5,662.85	0.287
			μ	853.85	0.043
TWIiS	10m	2,191	Z_{18}	816.25	0.372
			$\theta_{8,4}$	4,934.26	2.25
			μ	805.32	0.368

7.3.1.3 Asymmetric main and symmetric interaction effects

When considering the model where the main effects are asymmetric and the interaction effects are symmetric we assume that each of the two independent categorical variables comes from a different distribution and that only the effect of the interaction γ_{jk} exists, with $j \leq k$. Implementationally the effect γ_{kj} is set to be the same as the effect of the interaction γ_{jk}. Recalling the model for this case from 3.3.2.3 it is characterised as:

Model

$X_{ijk}|\mu, \alpha_j, \beta_k, I_{jk}, \theta_{jk}, \tau \sim \mathrm{N}(\mu + \alpha_j + \beta_k + I_{jk}\theta_{jk},\, \tau^{-1})$,

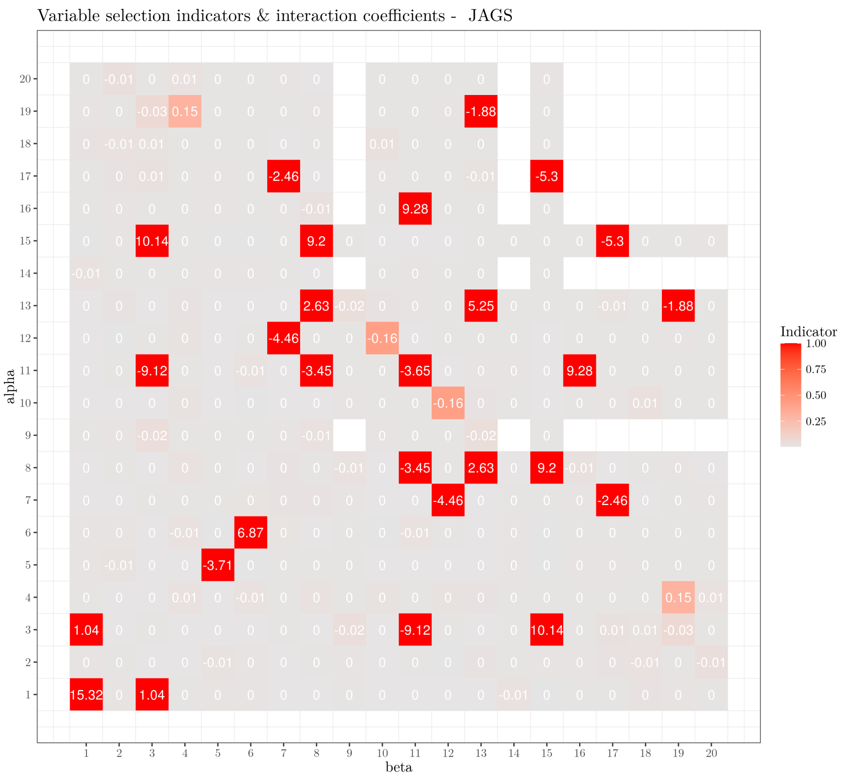
where $j \leq k$ for the indices of the interaction effects to impose symmetry, i=1,...,N, and N is the number of observations.

Results

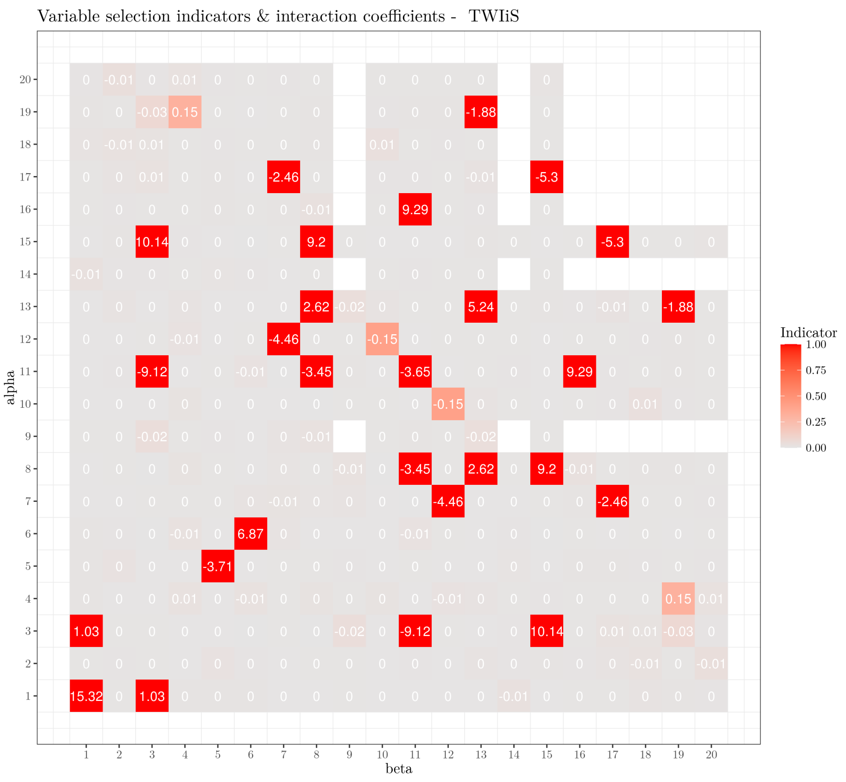
Following the same logic in presenting the results as for the previous models, first we depict the interaction terms in the form of heatmaps in Figure 7.11, to show the resemblance of the results from both programs to the true expected coefficients. The white cells of the graphs represent interactions for combinations of levels of the two categorical variables for which there are no observations in the dataset. In Table 7.5 we can see that, again as in the previous models, even though the main effects and μ estimated from both programs do not approximate the true values, the group means are very close to the expected.

In order to assess the convergence of the MCMC runs for this case we examine both the traceplots and the autocorrelation plots. An indicative illustration of two of these plots is shown in Figure 7.12. The traceplots display good mixing of the Gibbs sampler for both programs, proving that they explored well the posterior space. The autocorrelation plots have a decreasing tendency as the lag increases showing that the Gibbs sampler has converged to the target distribution. It is equally important for the validity of the results, to overlay the posterior densities from TWIiS and JAGS and check if they overlap, proving that they produce similar posterior draws. Figure 7.13 illustrates that this is valid for the variables randomly chosen for this example.

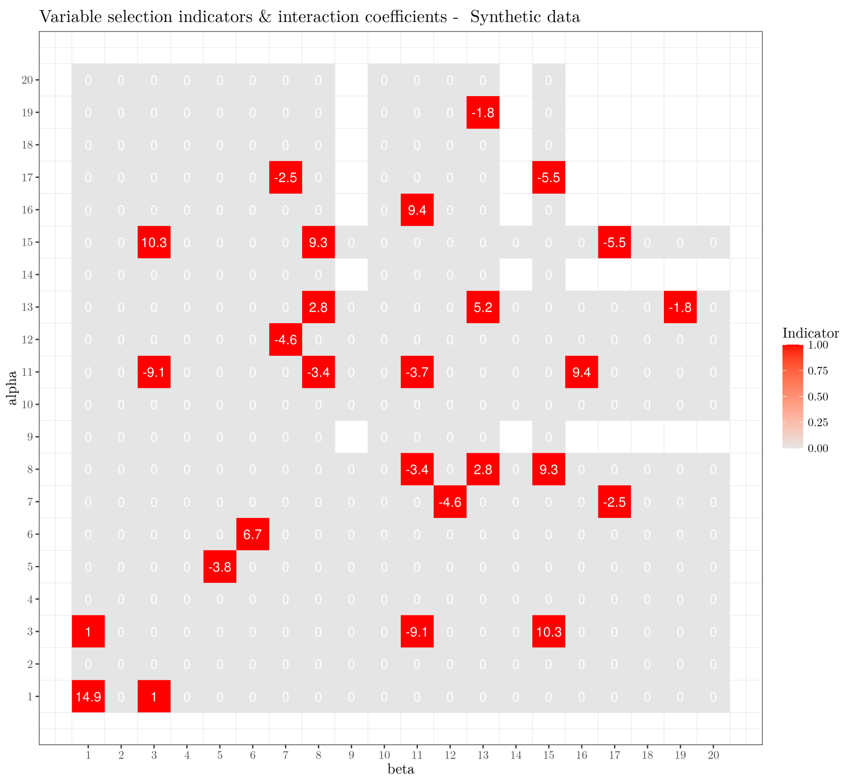
Finally, to compare the efficiency of the two programs examined and infer which shows better performance, we measure the runtime and assess the ESS and the ESS/sec. Table 7.6 summarises these metrics and we observe that, as in the previous two cases (7.3.1.1 and 7.3.1.2), the the ESS/sec is approximately seven times higher for TWIiS compared to JAGS, leading to the conclusion that the code written in Scala is more efficient than the equivalent JAGS program for modelling the asymmetric main and symmetric interaction effects approach using a Gibbs sampler.



(a) JAGS



(b) TWIiS

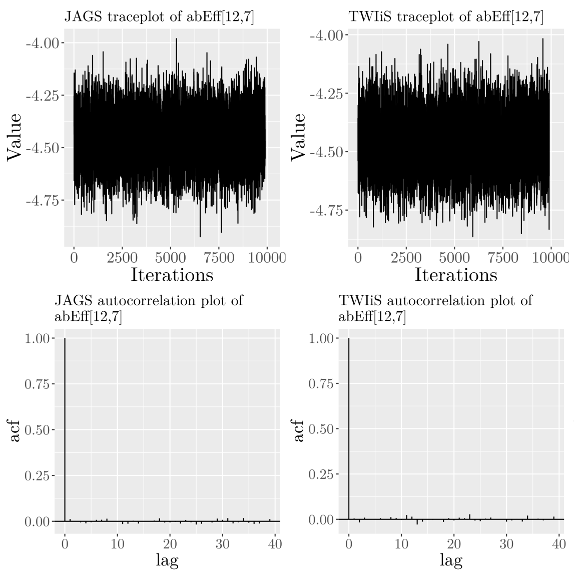


(c) True coefficients

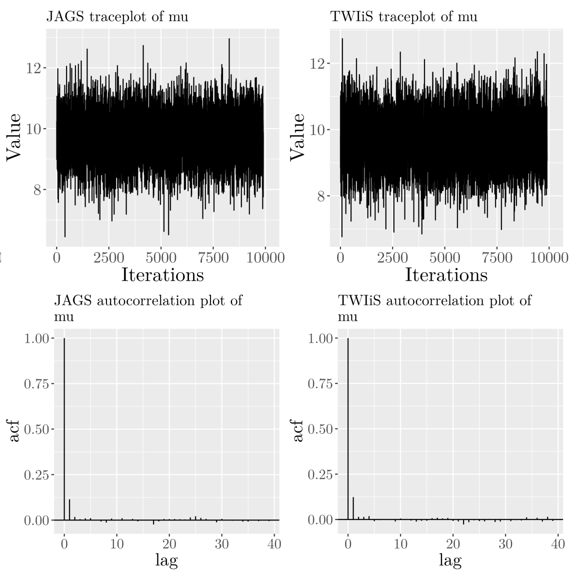
Figure 7.11: Comparison of the results among JAGS, TWIiS and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable.
Case: Asymmetric main & symmetric interaction effects

Table 7.5: Comparison of some effects and group means among JAGS, TWIiS and the true simulated coefficients.
Case: Asymmetric main & symmetric interaction effects

Variable	JAGS	TWIiS	Simulated
α_{12}	-0.960	-0.957	0.8
α_{15}	-0.240	-0.237	1.6
β_3	-0.452	-0.448	4.3
β_7	1.272	1.275	6.0
$\gamma_{12,7}$	-4.457	-4.457	-4.6
$\gamma_{15,3}$	10.139	10.138	10.3
μ	9.702	9.697	3.1
τ_α	1.159	1.146	2.0
τ_β	0.126	0.126	5.3
$\mu + \alpha_{12} + \beta_7 + \gamma_{12,7}$	5.557	5.558	5.3
$\mu + \alpha_{15} + \beta_3 + \gamma_{15,3}$	19.149	19.150	19.3



(a) Comparison - effect $\gamma_{12,7}$

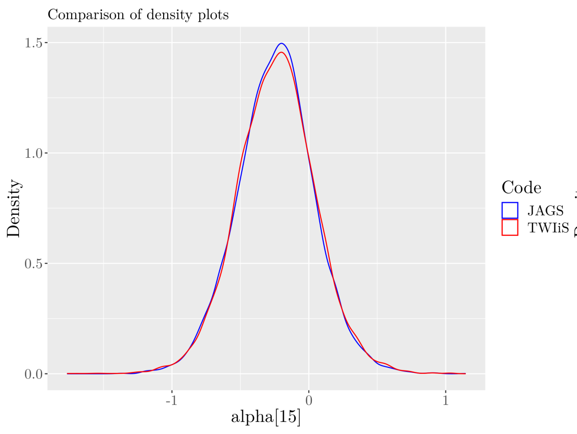


(b) Comparison - μ

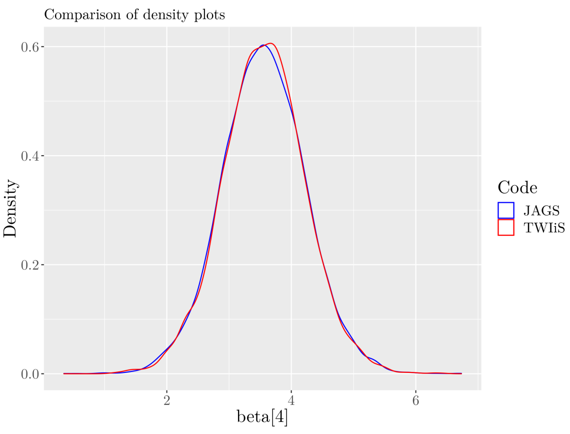
Figure 7.12: Comparison of traceplots and autocorrelation plots between JAGS and TWIiS. *Case: Asymmetric main & symmetric interaction effects*

Table 7.6: Efficiency measurement and comparison for Example 1: 15×20. *Case: Asymmetric main & symmetric interaction effects*

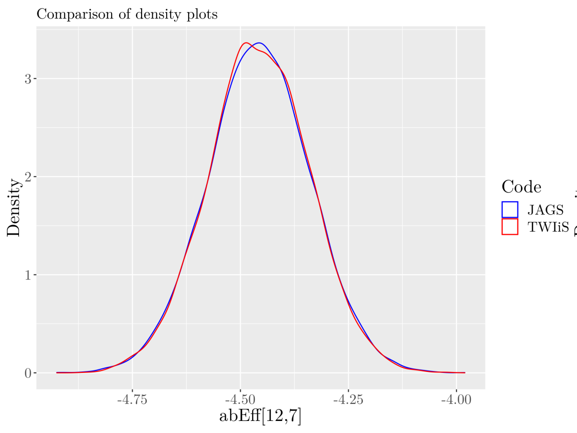
	Iterations	Time (secs)	Variable	ESS	ESS/sec
JAGS	1m	2,243	α_{13}	445.25	0.199
			β_7	108.38	0.048
			$\gamma_{1,6}$	3,968.76	1.769
			μ	106.29	0.047
TWIiS	1m	297	α_{11}	480.64	1.618
			β_9	92.89	0.312
			$\gamma_{12,7}$	9,900.00	2.656
			μ	94.20	0.317
JAGS	10m	20,901.6	α_5	3,942.14	0.189
			β_6	765.35	0.036
			$\gamma_{17,5}$	4,256.12	0.204
			μ	805.38	0.026
TWIiS	10m	2,608.9	α_{15}	3,859.52.00	1.479
			β_3	749.21	0.287
			$\gamma_{1,11}$	4,809.89	1.844
			μ	697.21	0.267



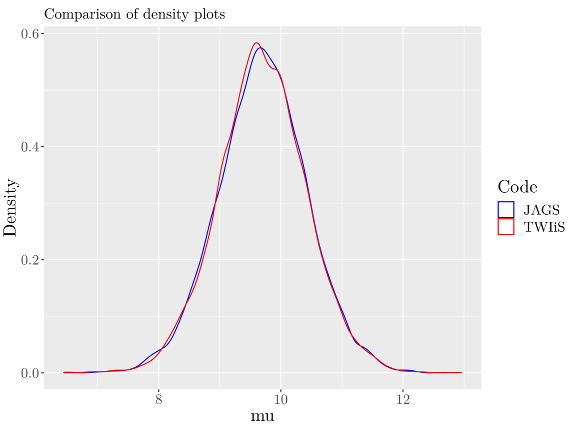
(a) Overlaid densities - effect α_{15}



(b) Overlaid densities - effect β_4



(c) Overlaid densities - interaction effect $\gamma_{12,7}$



(d) Overlaid densities - μ

Figure 7.13: Posterior density plots from JAGS and TWIiS.
Case: Asymmetric main & symmetric interaction effects

7.3.1.4 Symmetric main and symmetric interaction effects

The last case that we examine is a combination of the previous two models (7.3.1.2 and 7.3.1.3) and treats both the main and interaction effects as symmetric. The model for this case was presented in Section 3.3.2.3 and is specified as:

Model

$X_{ijk}|\mu, z_j, z_k, I_{jk}, \theta_{jk}, \tau \sim \mathrm{N}(\mu + z_j + z_k + I_{jk}\theta_{jk},\ \tau^{-1}),$

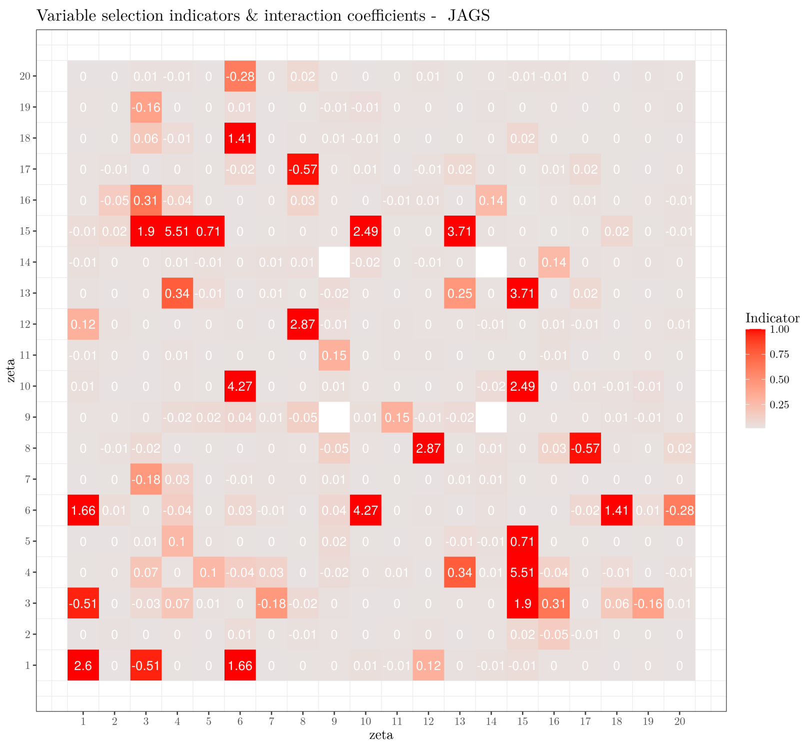
where $j \leq k$ for the indices of the interaction effects to impose symmetry, i=1,...,N, and N is the number of observations.

Results

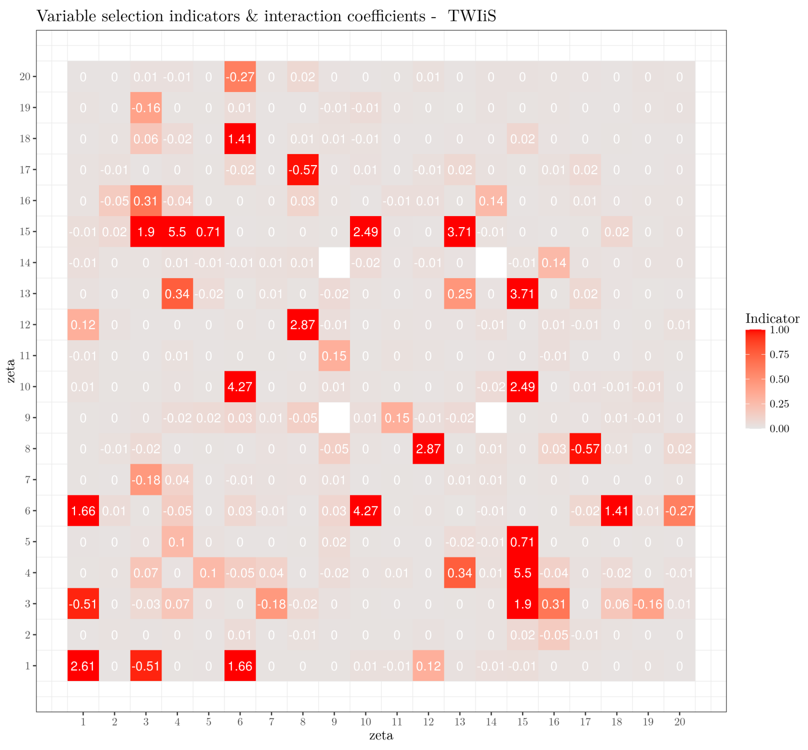
Similarly to the previous models, we can see that the values of the estimated interaction coefficients, from both TWIiS and JAGS, are close to the true values and the same interactions are considered important. The corresponding heatmaps illustrating these similarities are displayed in Figure 7.14. Regarding the mixing and convergence of the MCMC runs, the traceplots illustrated in Figure 7.15 show that both samplers have converged and adequately explored the posterior parameter space. Furthermore, the decreasing autocorrelation shows that the MCMC sample provides a more precise posterior estimate.

As in the models analysed earlier the same identifiability issue is encountered in this case as well; even though the main effects and μ estimated from TWIiS and JAGS are close, as illustrated in Figure 7.16, they differ from the true values of the simulated dataset. The same cannot be stated for the interaction effects estimated from both programs that are indeed similar to the true expected values. These observations are shown in Table 7.7. The similarity of the posterior distributions from the TWIiS and the JAGS code is apparent when overlaying the resulting posterior densities as presented in Figure 7.17.

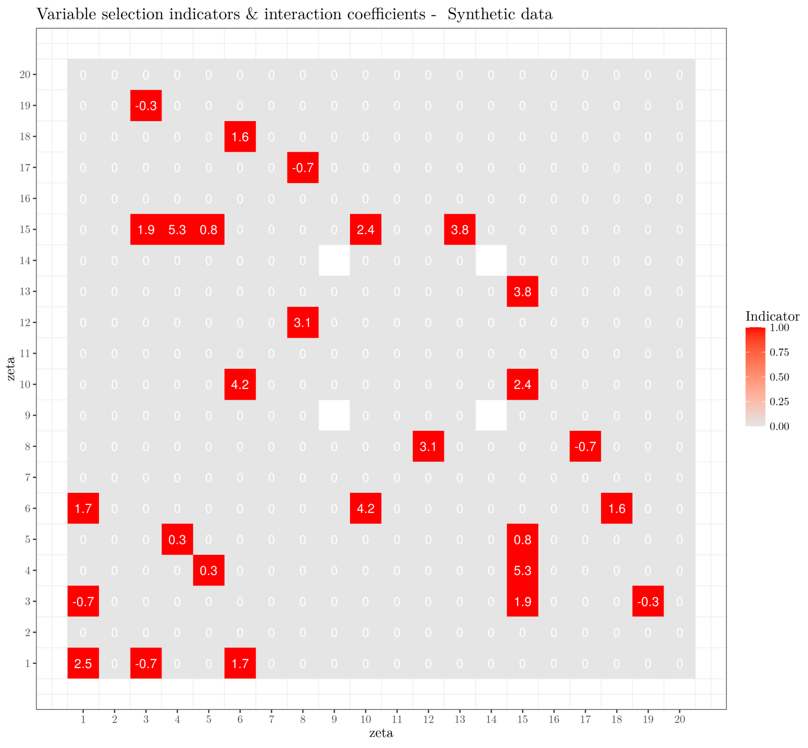
Finally, regarding the efficiency measures that allow us to compare the speed and effectiveness of the two programs, in Table 7.8 we can observe that even though the ESS for both programs is not significantly different, the smallest ESS/sec is better for TWIiS. In fact, it is approximately six times higher than the equivalent value for JAGS, suggesting that the Gibbs sampler implemented in Scala produces more independent samples per second and therefore has better performance than Jags.



(a) JAGS

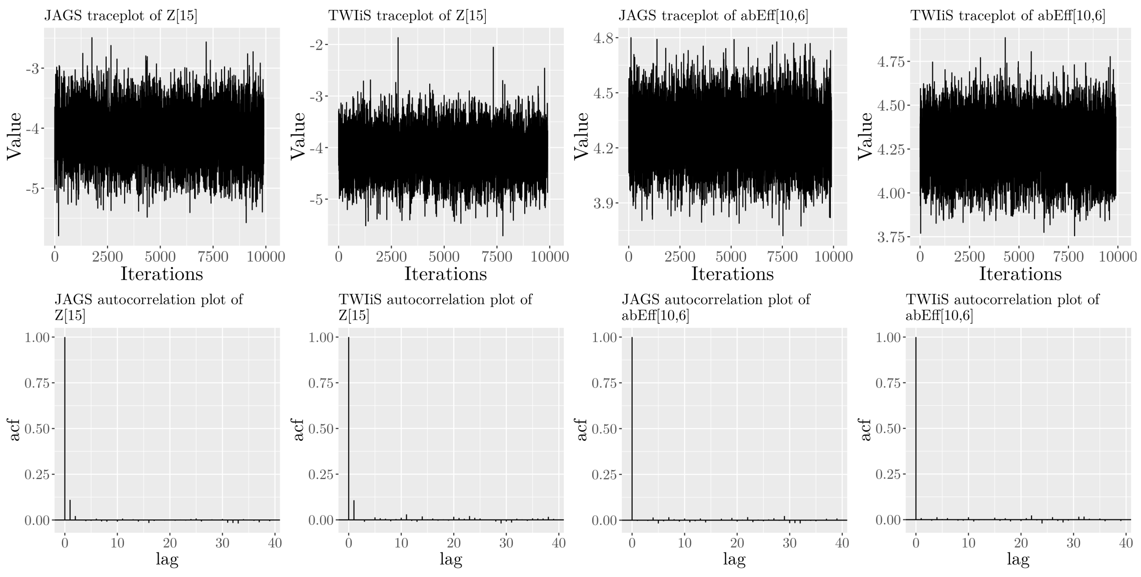


(b) TWIiS



(c) True coefficients

Figure 7.14: Comparison of the results among JAGS, TWIiS and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable.
Case: Symmetric main & interaction effects



(a) Comparison - effect Z_{15} (b) Comparison - effect $\gamma_{10,6}$

Figure 7.15: Comparison of traceplots and autocorrelation plots between JAGS and TWIiS. *Case: Symmetric main & interaction effects*

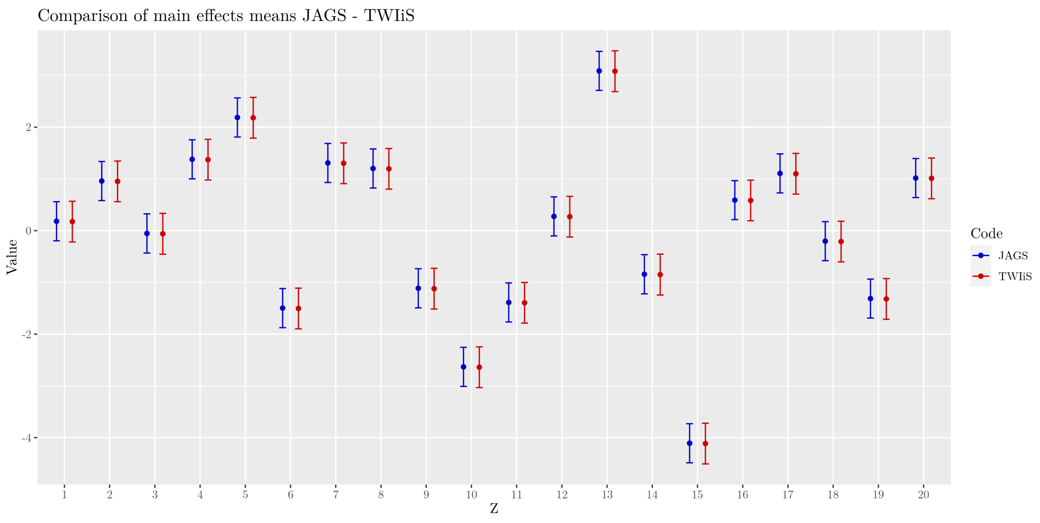
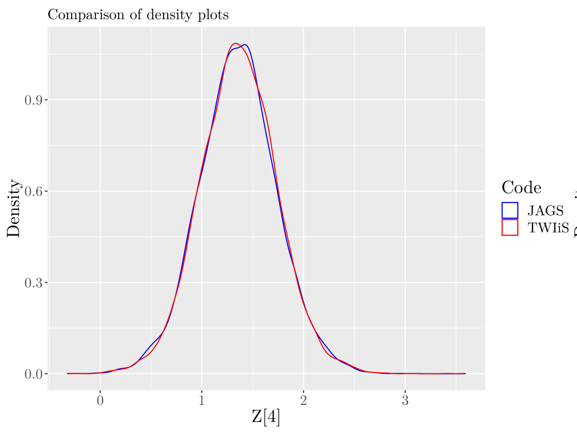


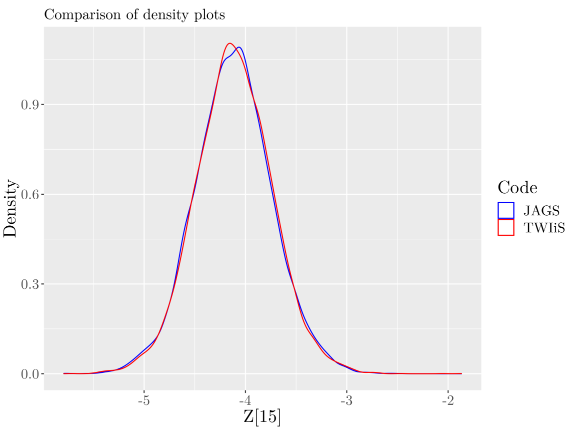
Figure 7.16: Comparison of main effects means between JAGS and TWIiS. *Case: Symmetric main & interaction effects*

Table 7.7: Comparison of some effects and group means among JAGS, TWIiS and the true simulated coefficients.
Case: Symmetric main & interaction effects

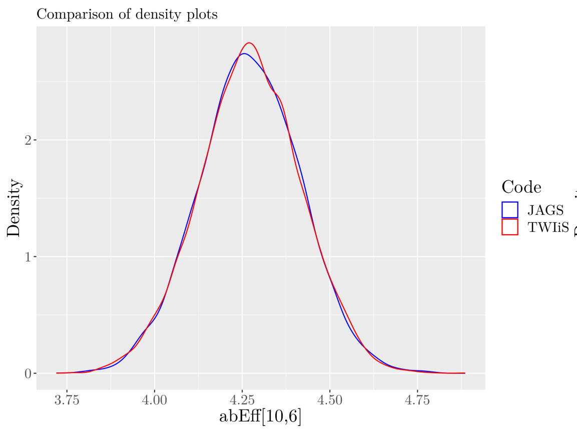
	JAGS	TWIiS	Simulated
z_1	0.174	0.178	1.3
z_6	-1.498	-1.492	-0.4
z_{10}	-2.639	-2.634	-1.5
$\gamma_{6,1}$	1.657	1.658	1.7
$\gamma_{10,6}$	4.273	4.273	4.2
μ	5.341	5.330	3.1
τ_z	0.387	0.389	3.5
$\mu + z_1 + z_6 + \gamma_{1,6}$	5.674	5.674	5.7
$\mu + z_6 + z_{10} + \gamma_{6,10}$	5.477	5.477	5.4



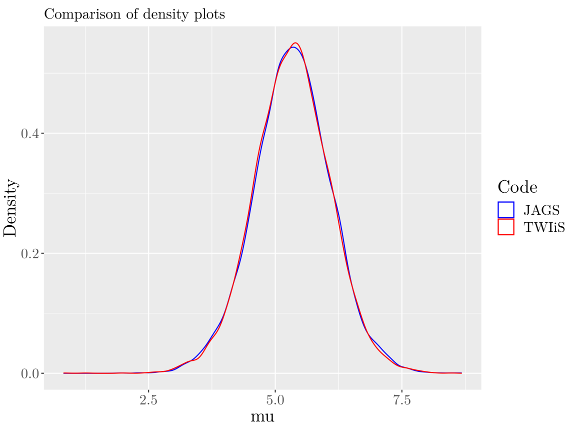
(a) Overlaid densities - effect Z_4



(b) Overlaid densities - effect Z_{15}



(c) Overlaid densities - interaction effect $\gamma_{10,6}$



(d) Overlaid densities - μ

Figure 7.17: Posterior density plots from JAGS and TWIiS.
Case: Symmetric main & interaction effects

Table 7.8: Efficiency measurement and comparison for Example 1: 15×20.
Case: Symmetric main & interaction effects

	Iterations	Time (secs)	Variable	ESS	ESS/sec
JAGS	1m	1,928	Z_{20}	82.7	0.043
			$\gamma_{18,4}$	4,508.47	2.34
			μ	88.06	0.046
TWIiS	1m	278	Z_5	85.94	0.309
			$\gamma_{1,8}$	4,606.68	16.57
			μ	95.84	0.344
JAGS	10m	18,928.7	Z_7	880.085	0.046
			$\gamma_{1,4}$	4,144.07	0.219
			μ	865.17	0.046
TWIiS	10m	3,699.5	Z_7	747.19	0.201
			$\gamma_{1,2}$	4,625.95	1.250
			μ	727.76	0.196

7.3.1.5 Summary of efficiency results for Example 1: 15×20

A collective presentation of the performance metric ESS/sec for all four model cases studied so far for both implementations is available in Table 7.9. Comparing the smallest values for each case between JAGS and TWIiS shows that TWIiS appears to be more efficient in a range that fluctuates between six and ten times depending on the model examined. The least efficiency gain appears in the case where both main and interaction effects are symmetric, whereas the best TWIiS performance compared to JAGS is for the case where the main effects are treated as symmetric. Finally, we can observe that both programs appear to draw more independent samples for the interaction effects compared to other estimated variables.

Table 7.9: Efficiency summary and comparison between JAGS and TWIiS for Example 1: 15×20, using variable selection indicators.

Code	Variable	ESS/sec			
		Both Asymmetric	*Symmetric Main*	*Symmetric Interactions*	*Both Symmetric*
		(2193 sec)	(2018 sec)	(2243 sec)	(1928 sec)
JAGS	γs	2.2	2.92	1.77	2.34
	μ	0.03	0.04	0.05	0.05
	αs	0.22	-	0.2	-
	βs	0.03	-	0.05	-
	zs	-	0.04	-	0.04

		(242 sec)	(220 sec)	(297 sec)	(278 sec)
TWIiS	γs	23.5	26	15	16.5
	μ	0.29	0.5	0.31	0.35
	αs	1.87	-	1.62	-
	βs	0.33	-	0.31	-
	zs	-	0.4	-	0.31

7.3.2 Example 2: Synthetic dataset with 50×60 levels

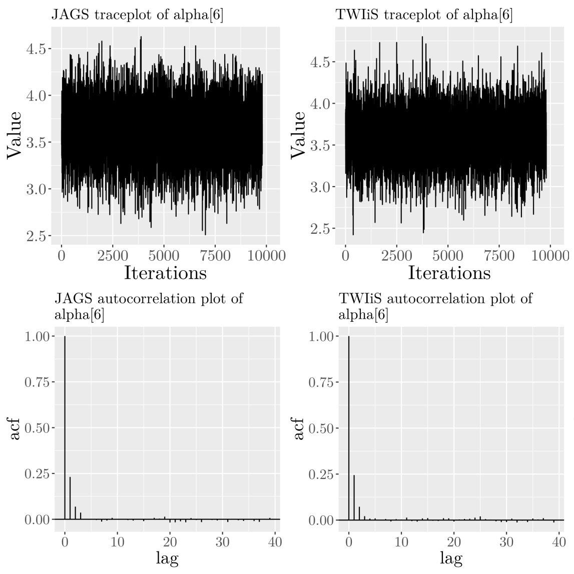
In order to verify the proper function and efficiency of the methods developed for variable selection, as well as the correctness of the results, it is necessary to extend the study and build a more complex model. Therefore, the code for both TWIiS and JAGS was tested on a bigger synthetic dataset including again two categorical variables, with 50 levels for the first and 60 levels for the second in the cases that involve asymmetric main effects, and 60 levels for both main effects in the cases where they are treated as symmetric. Similarly to the first example (7.3.1), variable selection using the Kuo and Mallick approach is applied to all four combinations of asymmetry in the effects.

To avoid repetition of similar findings we will provide a more analytical presentation of the results derived from the case where both the main and the interaction effects are asymmetric and we will selectively demonstrate the outcome of the remaining cases. For all cases the effective sample size and efficiency measures derive from one million iterations with a thinning factor of a hundred and further subsampling of the first 7,000 samples after achieving convergence, whilst retaining high autocorrelation values for the main effects and μ. For inference, runs of ten million iterations with respective adaptation of the thinning factor and an additional burn-in period which led to 9,000 samples were used.

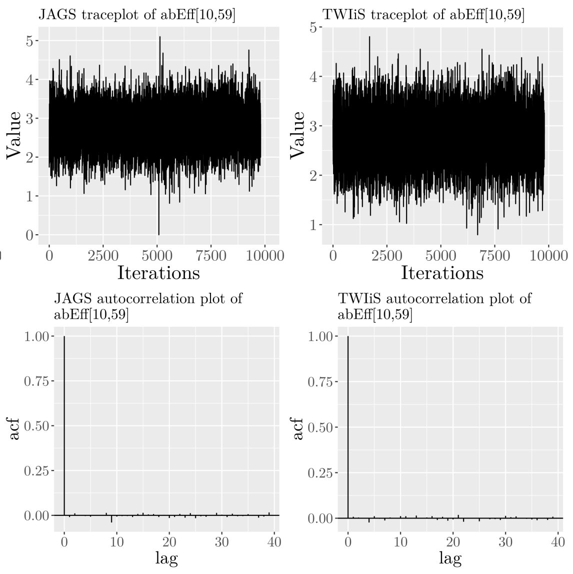
7.3.2.1 Asymmetric main and asymmetric interaction effects

The unknown variables in this case include the main effects α (50 levels) and β (60 levels) as well as the interaction coefficients γ_{jk} which are the product of the variable selection indicators and the strength of the interaction estimates. Furthermore, μ and τ which are the overall mean and precision, and τ_a, τ_b and τ_γ which represent the precision of the two main effects and the interactions respectively, are also unknown and have to be estimated.

The first step of the posterior analysis is to check the mixing and the autocorrelation for all the estimated variables. The diagnostic plots presented good mixing and decreasing autocorrelations for the estimated parameters suggesting that both algorithms explored sufficiently the posterior space and that the MCMC chains of both programs converged to the target distribution. Indicatively, there is an illustration of the traceplots and the autocorrelation plots for one main and one interaction effect in Figure 7.18.



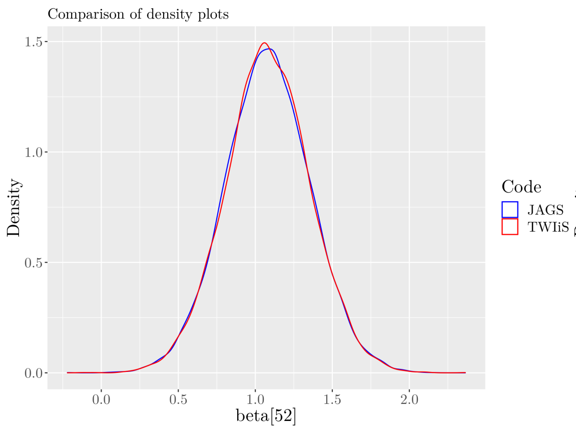
(a) Comparison - effect α_6



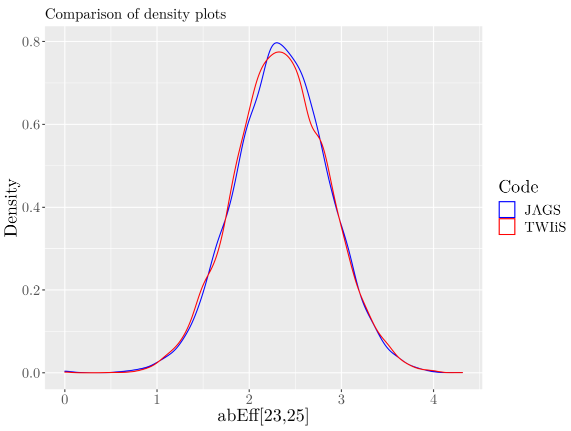
(b) Comparison - effect $\gamma_{10,59}$

Figure 7.18: Comparison of traceplots and autocorrelation plots from JAGS and TWIiS.
Case: Aymmetric main & interaction effects. Example 2: 50×60

Overlaying the posterior densities also showed that both pieces of code produce similar posterior distributions. Figure 7.19 illustrates the estimated posterior densities of the two programs for two randomly chosen variables.



(a) Comparison - effect β_{52}



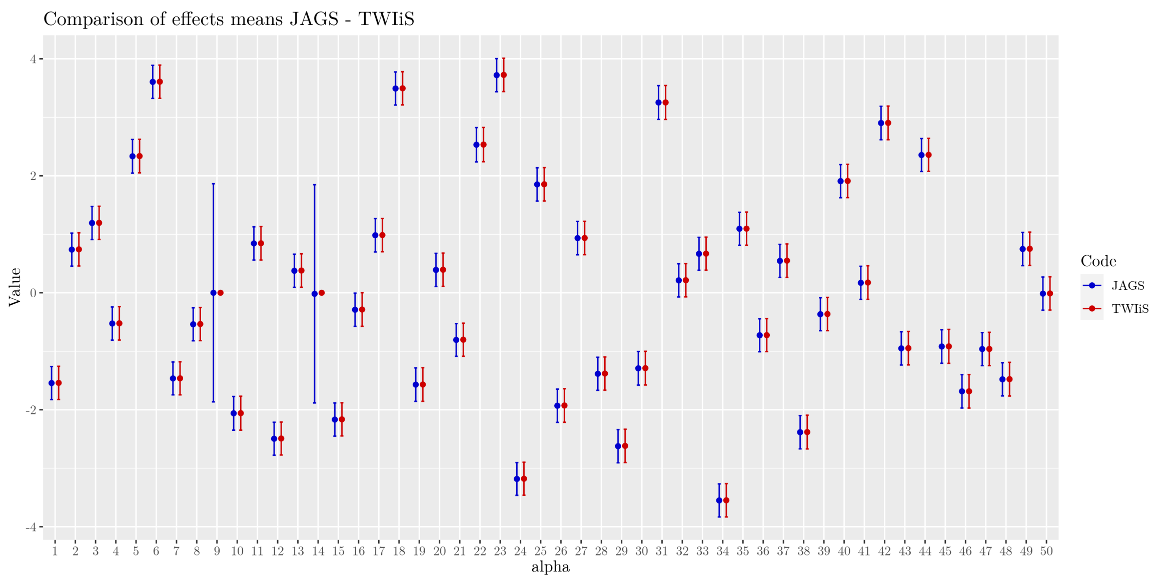
(b) Comparison - effect $\gamma_{23,25}$

Figure 7.19: Posterior density plots from JAGS and TWIiS.
Case: Asymmetric main & interaction effects. Example 2: 50×60

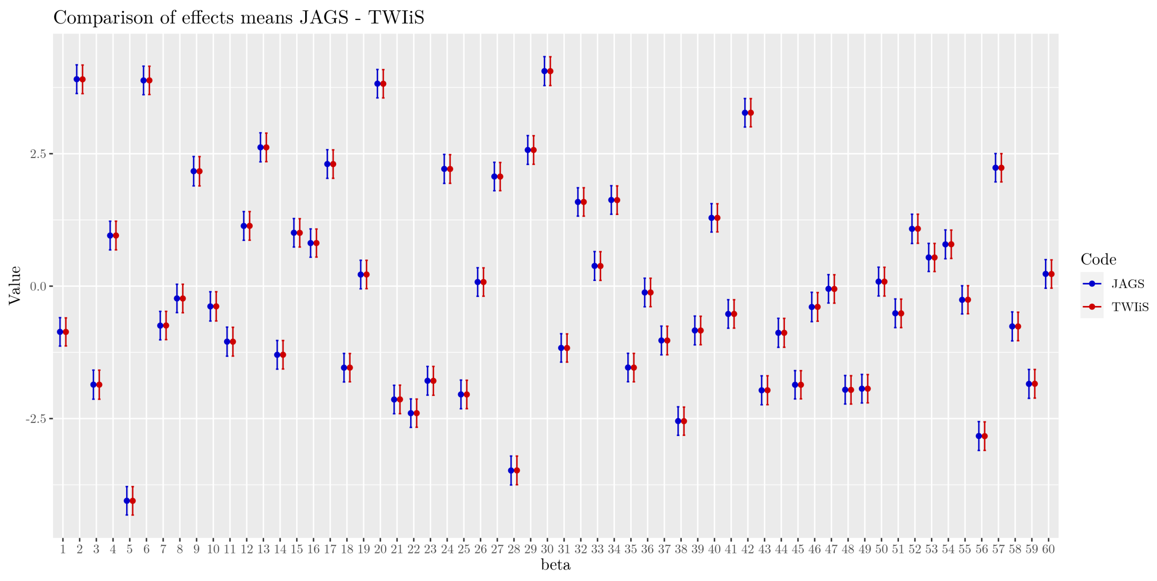
The comparison of the expectations for the main effects α and β estimated from both programs are presented in Figure 7.20. They are very similar, with the exception of the effects α_9 and α_{14} due to the different way that the two programs handle missing levels. As mentioned earlier, JAGS samples from their prior, whereas TWIiS, since the levels do not exist, sets their value to zero. The expected coefficients are not depicted in these plots because similarly to the previous example with fewer levels, an identifiability issue arose for the main effects. Therefore, the true coefficients of the synthetic dataset for the main effects, do not correspond to the ones estimated from JAGS and TWIiS. Nevertheless, there is accordance between the group means of the estimated and the expected coefficients that will be illustrated later in this section.

Since the number of interactions in this example is large (3,000), heatmaps constitute a useful tool to compare the interaction effects estimated from JAGS and TWIiS. Firstly, Figure 7.21 displays a heatmap presenting the strength of the variable selection indicators. Secondly, the heatmap in Figure 7.22 depicts the strength of the interaction effects. In both cases, the heatmaps illustrating the results from TWIiS and JAGS follow very similar patterns. This is also accurate when we examine the equivalent heatmap of the true coefficient values of the synthetic dataset. In all heatmaps the cells for the interactions with no relevant observations in the dataset are denoted with white. Even though these heatmaps emphasise the general pattern followed rather than individual estimates, it can be noticed that for some interactions, the indicator heatmaps for both JAGS and TWIiS suggest a low inclusion probability, whereas the equivalent true coefficient shows that this interaction was simulated to be included in the model. This is due to the fact that the corresponding strength of the interaction is low. Therefore, for some low (in absolute value) values of interaction coefficients JAGS and TWIiS usually assign lower inclusion probabilities.

As mentioned earlier in this section, the identifiability problem results in different estimated values for the main effects and the mean, than the expected ones. The third type of heatmap (Figure 7.23) aims to depict the proximity of the results regarding the group means derived from JAGS and TWIiS to the true group means.

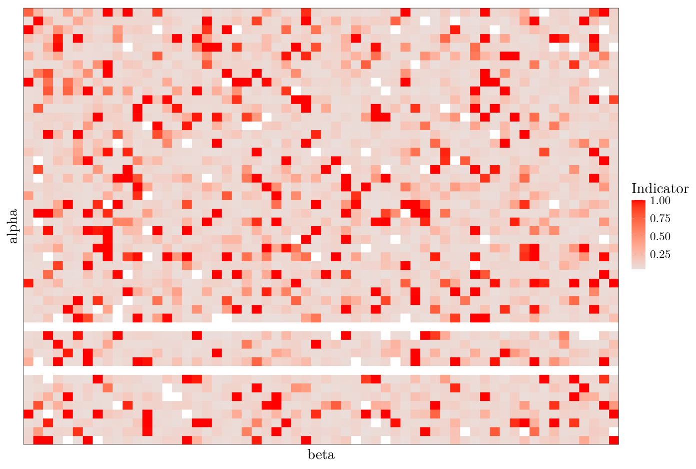


(a) Comparison - αs

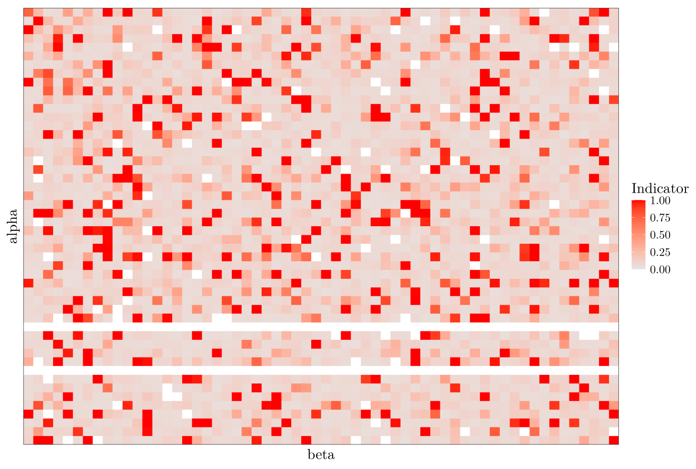


(b) Comparison - βs

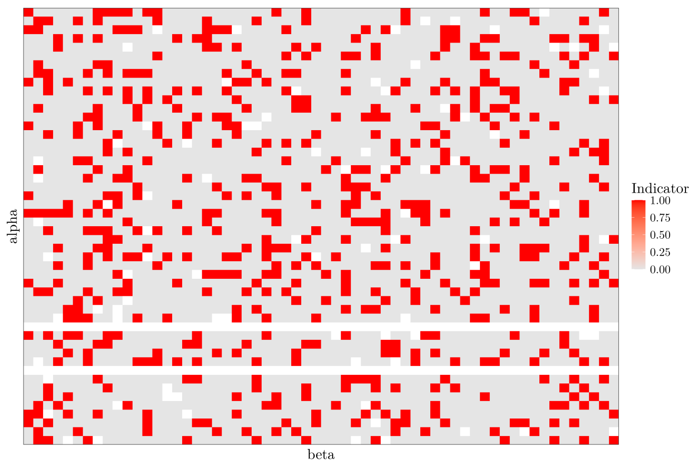
Figure 7.20: Comparison of main effects means estimated from JAGS and TWIiS.
Case: Asymmetric main & interaction effects. Example 2: 50×60



(a) JAGS

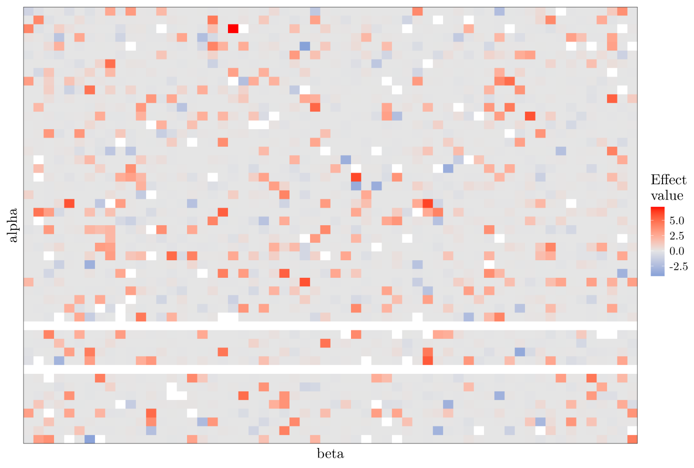


(b) TWIiS

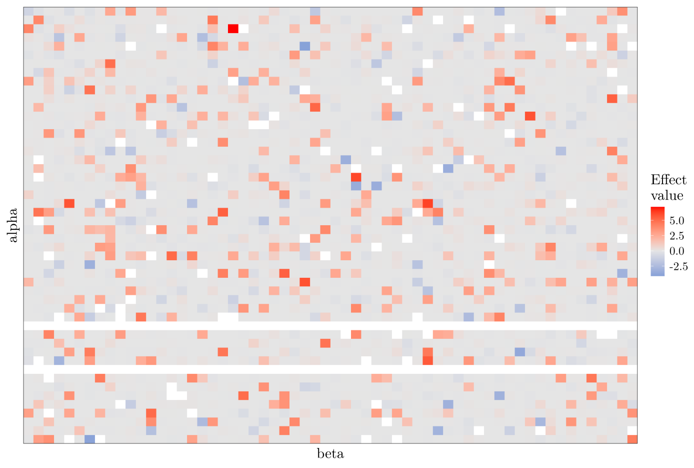


(c) True coefficients

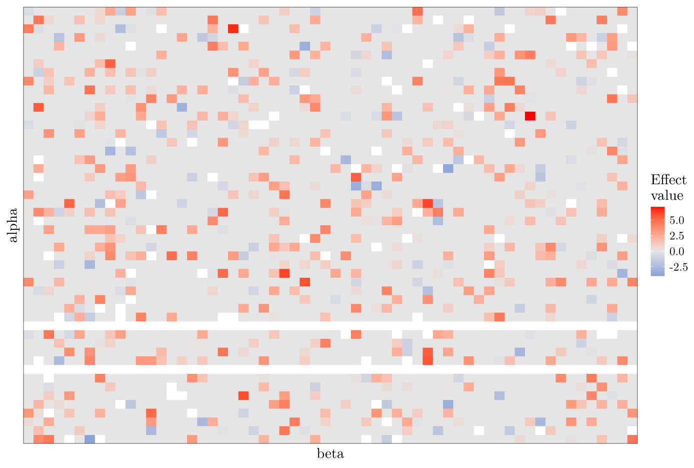
Figure 7.21: Comparison of the strength of indicators from JAGS and TWIiS, and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable.
Case: Asymmetric main & interaction effects. Example 2: 50×60



(a) JAGS

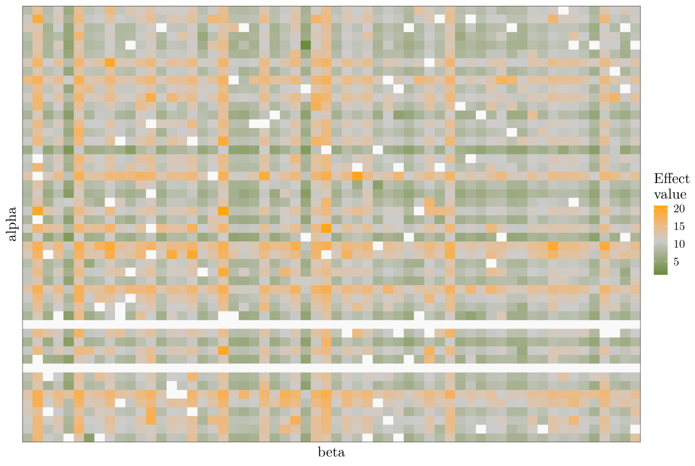


(b) TWIiS

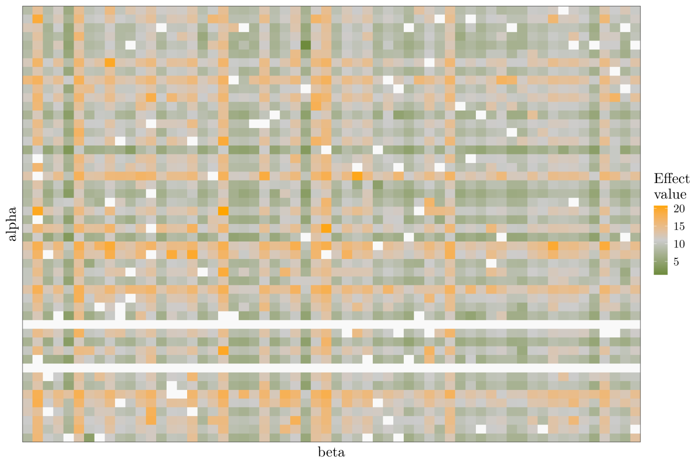


(c) True coefficients

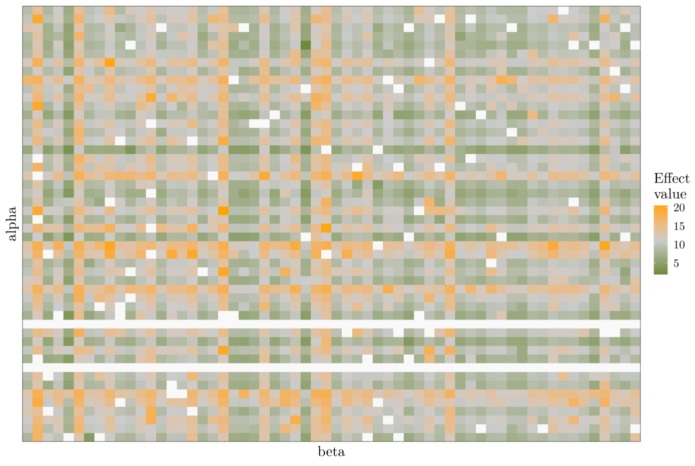
Figure 7.22: Comparison of the strength of interaction coefficients among JAGS, TWIiS and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable.
Case: Asymmetric main & interaction effects. Example 2: 50×60



(a) JAGS



(b) TWIiS

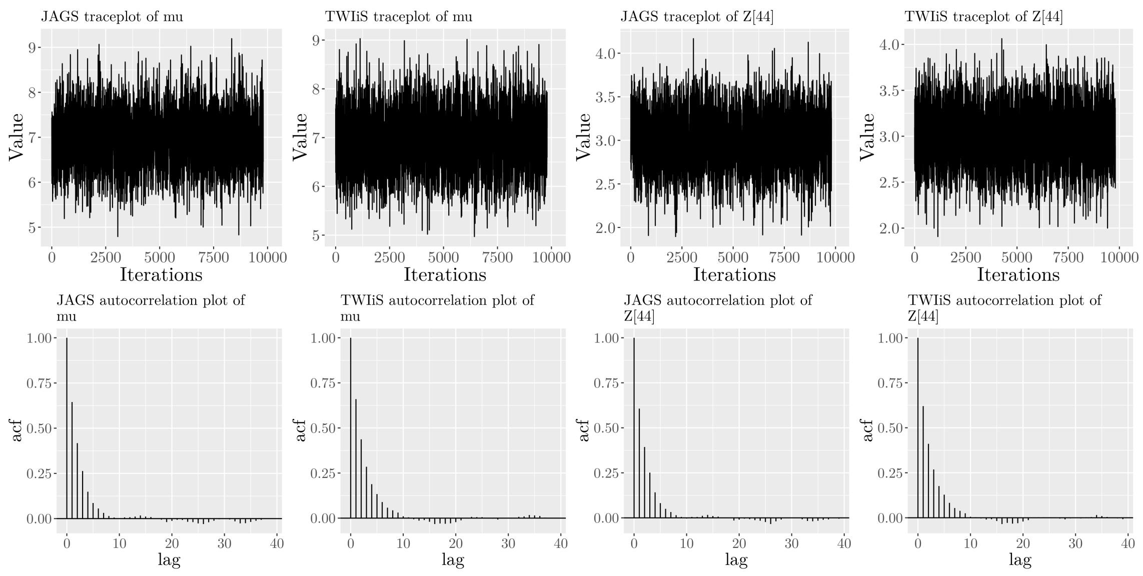


(c) True coefficients

Figure 7.23: Comparison of the group means among JAGS, TWIiS and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable.
Case: Asymmetric main & interaction effects. Example 2: 50×60

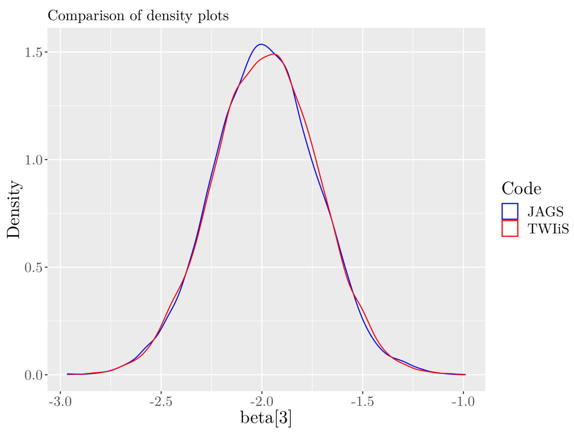
7.3.2.2 Remaining cases of Example 2: 50×60

The remaining cases that were explored for this dataset are models that treat the main effects only as symmetric, the interaction effects as symmetric and their combination. Overall the outcome of these models follows a similar logic as the results presented for the previous case. The diagnostic plots show that the posterior samples generated from both MCMC implementations are sufficient to provide accurate approximations of the target distribution. The traceplots do not include flat regions and indicate good mixing. In addition, the autocorrelation plots show decaying dependence between the samples as the lag increases. Samples of some indicative diagnostic plots for the case where the main effects are treated as symmetric are available in Figure 7.24. The posterior densities produced by JAGS and TWIiS overlap in all cases proving their similarity as shown in Figure 7.25 for some parameters estimated from the model involving symmetric interactions. The identifiability issue is still present in all cases. Consistent with the findings so far is also the identification of the same interactions as important and the calculation of the group means that shows proximity between the results from JAGS, TWIiS and the true coefficients as illustrated in the heatmaps presented in Figure 7.26 and with regard to the case where both the main and the interaction effects are symmetric.

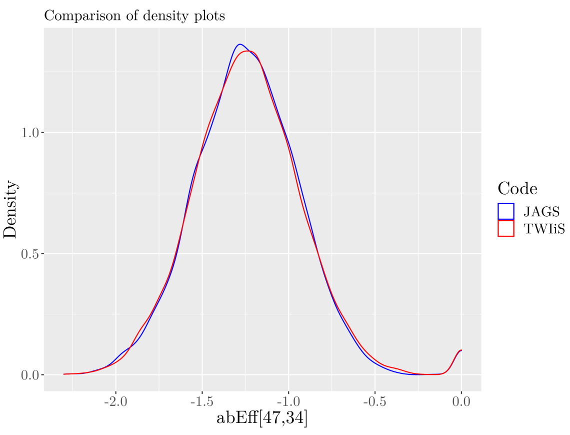


(a) Comparison - μ (b) Comparison - effect Z_{44}

Figure 7.24: Comparison of traceplots and autocorrelation plots from JAGS and TWIiS. *Case: Symmetric main & asymmetric interaction effects. Example 2: 60×60*

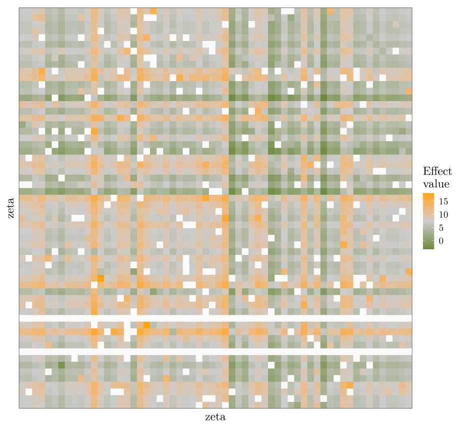


(a) Comparison - effect β_3

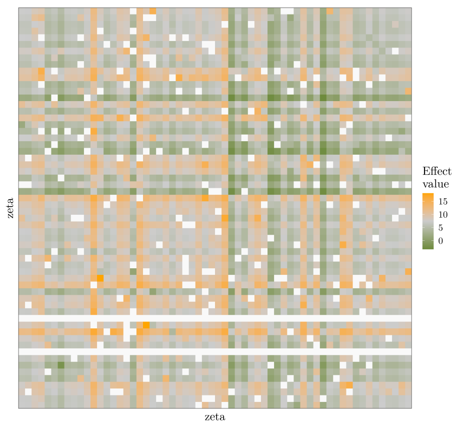


(b) Comparison - effect $\gamma_{47,34}$

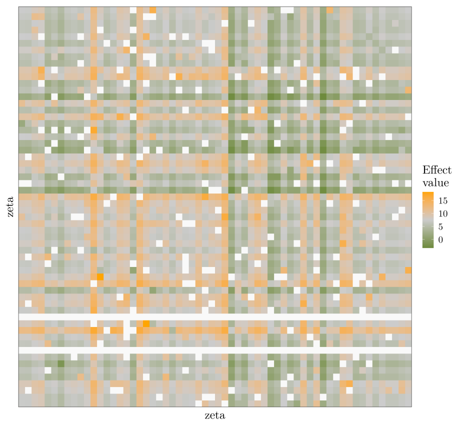
Figure 7.25: Comparison of overlaid densities from JAGS and TWIiS.
Case: Asymmetric main & symmetric interaction effects. Example 2: 50×60



(a) JAGS



(b) TWIiS



(c) True coefficients

Figure 7.26: Comparison of the group means among JAGS, TWIiS and the true coefficients. The y-axis and the x-axis represent Zs.
Case: Symmetric main & interaction effects. Example 2: 60×60

7.3.2.3 Summary of efficiency results for Example 2: 50×60

Concerning the efficiency comparison between JAGS and TWIiS, Table 7.10 provides a summary of the efficiency metrics estimated from all models. It includes the smallest measurements for all groups of variables to emphasise how the trend is formed. For each case we compare the minimum ESS/sec value and we can see that TWIiS has better performance than JAGS in this example with the particular characteristics considered regarding the sample size and complexity of the model. The degree of performance improvement differs with the case. For models assuming the interactions as symmetric TWIiS appears to be 3.7 times more efficient than JAGS, whereas this ratio is approximately 2.4 for the case that treats both effects as symmetric. In addition, the Gibbs sampler developed in both programs appears to sample the interaction effects more efficiently than the main effects.

Table 7.10: Efficiency summary and comparison between JAGS and TWIiS for Example 2: 50×60, using variable selection indicators.

Code	Variable	ESS/sec			
		Both Asymmetric	*Symmetric Main*	*Symmetric Interactions*	*Both Symmetric*
JAGS		(3436 sec)	(4421 sec)	(2993 sec)	(3643 sec)
	γs	1.45	1.18	1.68	1.21
	μ	0.1	0.03	0.1	0.05
	αs	0.14	-	0.17	-
	βs	0.2	-	0.2	-
	zs	-	0.03	-	0.05
TWIiS		(912 sec)	(1583 sec)	(958 sec)	(1105 sec)
	γs	5.52	3.22	4.65	4.09
	μ	0.32	0.09	0.37	0.14
	αs	0.52	-	0.53	-
	βs	0.63	-	1.66	-
	zs	-	0.09	-	0.12

7.3.3 Example 3: Synthetic dataset with 113×143 levels

Further exploration of the behaviour of the two programs in more complex models was achieved with the use of two synthetic datasets with 113 levels for the first and 143 for the second categorical variable and varying sizes; the first dataset involves 10,000 and the second 80,000 observations. A model that treated both the main and interaction effects as symmetric was fitted for both examples. After assessing the MCMC diagnostics that were mentioned in the previous examples and confirming the similarity of the posterior estimates between JAGS and TWIiS, an efficiency comparison was made to understand how their performance is formed in this case. Table 7.11 summarises these results and shows that as in the previous examples TWIiS seems to be more efficient than the code written in JAGS. Looking at the smallest values of the ESS/sec for the example with 10,000 observations TWIiS appears to be 2.7 times more efficient than JAGS. This ratio is increased for the biggest dataset, where comparing the values for the means that have the smallest ESS/sec for both programs renders TWIiS five times more efficient than the equivalent JAGS code.

Table 7.11: Efficiency summary and comparison between JAGS and TWIiS for Example 3: 113×143, using variable selection indicators.
Case: Asymmetric main & interaction effects

Code	Variable	ESS/sec	
		10,000 observations	80,000 observations
		(8543 sec)	(51599 sec)
	γs & τs	0.54	0.076
JAGS	μ	0.14	0.004
	αs	0.22	0.008
	βs	0.39	0.005
		(2912 sec)	(7326 sec)
	γs & τs	1.7	0.54
TWIiS	μ	0.38	0.02
	αs	0.64	0.053
	βs	1.23	0.028

7.4 Variable selection using the Horseshoe prior

In the previous examples we used the Kuo and Mallick approach to identify the important interactions of the two-way Anova models that we examined. This is a classical approach in Bayesian variable selection for linear regression, as the embedded indicator variables, one for each predictor p, incorporate all possible 2^p submodels. The posterior likelihood of each indicator represents the probability of its corresponding predictor being included in the model and can be estimated using classical MCMC algorithms. One characteristic of the Kuo and Mallick approach is that it involves discrete variables. Gibbs samplers can target discrete distributions. Therefore, since JAGS uses this MCMC sampler, it was used for developing this variable selection method in R. Equivalently, the code for TWIiS was implemented in Scala sharing similar logic.

Another Monte Carlo algorithm, Hamiltonian Monte Carlo (HMC), explored in Section 2.2.3, uses information about the geometry of the posterior distribution through its gradient to propose new values. HMC can be more efficient than the Gibbs sampler, but due to the fact that the log-posterior needs to be differentiable everywhere, it does not allow inference for discrete parameters, especially in cases where it is not possible to analytically marginalise out the discrete variables. Consequently, HMC cannot be used in Bayesian variable selection using the indicator approach previously examined and implemented. However, variable selection using HMC can be achieved using the Horseshoe prior (Section 2.4.2), that depends only on continuous distributions and more precisely the half-Cauchy.

In this section we are going to explore the application of the Horseshoe prior for variable selection on the interactions of our two-way Anova model with random effects and various levels of asymmetry. This alternative implementation is embedded in JAGS and TWIiS by alterations in the code to associate Horseshoe priors to the categorical variables, with both programs still using a Gibbs sampler. Variable selection using Horseshoe priors along with HMC, is also considered and carried out in R using the probabilistic programming language Stan and in Scala using the library Rainier. Although the cost per iteration is greater for HMC than for other MCMC algorithms such as the Gibbs sampler, HMC is considered to generate proposals more efficiently and can show better mixing with fewer samples.

In the following sections, we are going to present and compare the results derived from the versions of JAGS and TWIiS that incorporate the Horseshoe prior, using the same synthetic datasets of the previous examples for the indicator approach (Section 7.3). Then, we will introduce the results obtained from Rainier and Stan using a smaller example and compare them to the equivalent results obtained from JAGS and TWIiS not only on the basis of posterior estimates but also in terms of efficiency. Finally, the last part aims to compare the implementations using the Horseshoe prior and the variable selection indicators and highlight the methodology that appears to be more appropriate and efficient for the particular models studied.

7.4.1 Horseshoe with JAGS and TWIiS

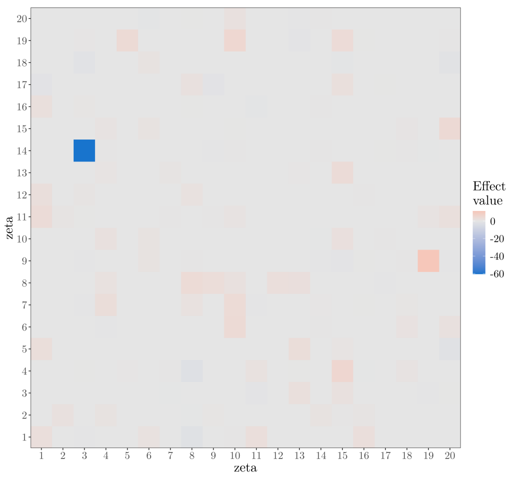
Recalling the intuition behind the Horseshoe prior presented in Section 2.4.2 and analytically embedded in a Gibbs sampler in Section 3.3.3, there exists a global parameter τ_{HS} that pulls all the weights of the interaction effects globally towards zero. In addition, for each interaction γ_{jk} there is an associated local parameter λ_{jk} that due to the thick tails of the half-Cauchy distribution that it follows, allows some of the weights to escape shrinkage.

The results and performance of JAGS and TWIiS implementing variable selection on the interactions using the Horseshoe prior were tested on three examples with synthetic datasets of 10,000 observations. As in the case of the Kuo and Mallick approach the model built in the first example involved 15 levels for the first and 20 levels for the second categorical variable, while the second example had 50 and 60 levels respectively. The third example included more levels and varying sample sizes. Both JAGS and TWIiS were assessed for all cases of asymmetry in the effects for the first two examples, whereas the third was applied only in the case where both the main and interaction effects are asymmetric. For producing the efficiency measurements the programs run for one million iterations with a thinning factor of a hundred. An extra burn-in period was added leading to 9,900 samples. For these runs the algorithms showed convergence whilst retaining autocorrelation between the samples. Longer runs of ten million iterations and proportional adaptation of the thinning factor were used for inference.

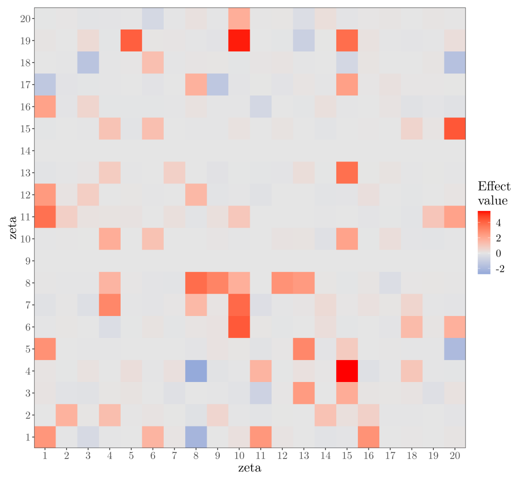
7.4.1.1 Example 1: Synthetic dataset with 15×20 levels

For the models that treated both the main and the interaction effects as asymmetric, only the interaction effects as symmetric and both main and interaction effects as symmetric, the results were similar to the equivalent estimates produced from the models using the Kuo and Mallick approach. The mixing was good for all the unknown parameters showing that the MCMC has explored the parameter space adequately. The autocorrelation plots showed low autocorrelation for the samples regarding the interaction effects and the precisions, and decreasing tendencies as the lag increased for the main effects and μ. The posterior densities estimated from both programs for the unknown variables of the model overlap indicating their equality. Comparison of the results regarding the main effects from both programs show that identifiability issues have arisen, however both programs return similar estimates for these values. Nevertheless, the group means produced from

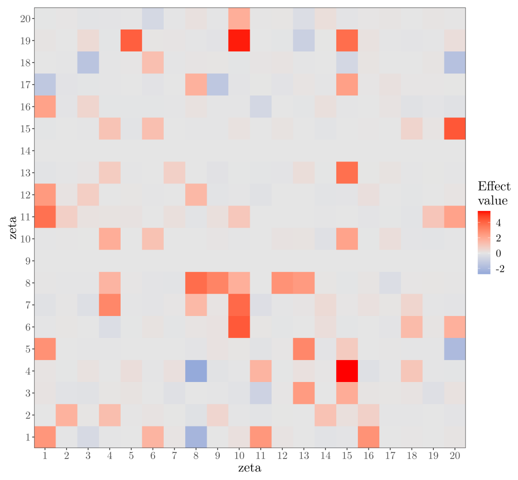
TWIiS were in close proximity to the equivalent estimates deriving from JAGS, as well as the true expected values. Samples of these plots for some randomly chosen variables can be found in Appendix A.1 (Figures A.1 and A.2). The main inaccuracy of the Horseshoe prior approach appears in the case where only the main effects are symmetric. Knowing that during the simulation process of the synthetic dataset some levels for the first categorical variable were intentionally omitted, it would be expected that the estimated coefficients of the interactions involving these levels would be close to zero. However, JAGS returns high estimated values for two interaction effects that fall in this category, and more precisely $\gamma_{9,19} = 11$ and $\gamma_{14,3} = -60$. The heatmaps presented in Figure 7.27 illustrate the estimated interaction coefficients for this case.



(a) JAGS before removing wrongly identified variables



(b) JAGS after removing wrongly identified variables



(c) TWIiS

Figure 7.27: Heatmaps of the interaction strengths using variable selection with the Horseshoe prior for the symmetric main effects case. Highlighted inaccuracies for $\gamma_{9,19}$ and $\gamma_{14,3}$ in (a). Values from JAGS corrected in (b) resulting to a heatmap similar to TWIiS (c).

In these heatmaps the cells regarding missing levels are not marked in white as in the cases where they were previously used in order to highlight the differences in this example.

Regarding the efficiency measurements in all four cases of asymmetry using the Horseshoe prior, Table 7.12 summarises the relevant values calculated. As in the analysis for the variable selection with indicators, the table presents the minimum ESS along with the variable for which it was observed and the corresponding ESS/sec for both programs based on the runtimes referred below. Calculating the ratio between the two programs for the various implementations indicates that TWIiS is approximately four to five times more efficient than the equivalent JAGS code for the model complexity and sample size considered in the example. Furthermore, similarly to the indicator approach, the Gibbs sampler with the Horseshoe prior draws samples more efficiently for the interaction effects than the other parameters.

Table 7.12: Efficiency summary and comparison between JAGS and TWIiS for Example 1: 15×20, using the Horseshoe prior.

Code	Variable	ESS/sec			
		Both Asymmetric	*Symmetric Main*	*Symmetric Interactions*	*Both Symmetric*
JAGS		(1699 sec)	(1815 sec)	(1559 sec)	(1542 sec)
	γs	2.19	3.2	3.21	3.08
	μ	0.05	0.05	0.03	0.06
	αs	0.31	-	0.24	-
	βs	0.06	-	0.03	-
	zs	-	0.05	-	0.06
TWIiS		(382 sec)	(378 sec)	(481 sec)	(342 sec)
	γs	15.4	15.7	11.35	17.55
	μ	0.25	0.22	0.13	0.24
	αs	1.8	-	1.08	-
	βs	0.26	-	0.15	-
	zs	-	0.22	-	0.21

7.4.1.2 Example 2: Synthetic dataset with 50×60 levels

Using a second example with more levels increases the complexity of the model and allows us to understand whether and how the results and efficiency are influenced. The models used for the synthetic dataset with 50 levels for the first and 60 levels for the second categorical variable result in similar estimates as using the Kuo and Mallick approach. The traceplots and the autocorrelation plots examined indicate that the Markov chain has converged to the target distribution after the algorithm has sufficiently explored the posterior space for all the unknown variables. Again, identifiability issues are observed regarding the estimates for the main effects, that even though similar for JAGS and TWIiS differ from the expected values. However, as in the previous examples the estimated group means are similar to the true group mean values. Both programs identify correctly the interaction coefficients that should be included in the model based on our knowledge about the simulation process of the synthetic dataset and the true values. An exception holds for the case where both the main and interaction effects are asymmetric. As in the previous smaller example, the model built using the Gibbs sampler along with the Horseshoe prior in JAGS wrongly identifies as important interactions that concern combinations of levels for which there are no observations. Samples of heatmaps, diagnostic plots and overlaying posterior density plots for this example are available in Appendix A.2.

Conclusions about the efficiency of the two programs can be drawn by looking at Table 7.13. TWIiS presents two to five times higher ESS/sec compared to JAGS showing that, similarly to the previous example, its implementation is more efficient than the more classic approach.

Table 7.13: Efficiency summary and comparison between JAGS and TWIiS for Example 2: 50×60, using the Horseshoe prior.

Code	Variable	ESS/sec			
		Both Asymmetric	*Symmetric Main*	*Symmetric Interactions*	*Both Symmetric*
		(7254 sec)	(8564 sec)	(4979 sec)	(4972 sec)
JAGS	γs	0.57	0.5	0.7	0.7
	μ	0.04	0.02	0.07	0.03
	αs	0.07	-	0.11	-
	βs	0.09	-	0.12	-
	zs	-	0.02	-	0.03
		(1835 sec)	(2075 sec)	(1773 sec)	(2016 sec)
TWIiS	γs	2.96	2.7	3.11	2.75
	μ	0.18	0.07	0.22	0.08
	αs	0.25	-	0.34	-
	βs	0.32	-	0.36	-
	zs	-	0.07	-	0.07

7.4.1.3 Example 3: Synthetic dataset with 113×143 levels

The last example tested was a model with augmented number of levels and two different sample sizes, to understand how an increase in the complexity and number of observations affects the performance of each program in the case of the Horseshoe prior approach. This example regards a model where both the main and interaction effects are considered to be asymmetric. After assessing all the necessary diagnostics the performance metrics were calculated and are summarised in Table 7.14. In the example with a sample size of 10,000 observations TWIiS appears to be six times more efficient than its equivalent JAGS code. This ratio increases to seven in the model that involved 80,000 observations, showing that it is likely for TWIiS to perform even better compared to JAGS in bigger datasets.

Table 7.14: Efficiency summary and comparison between JAGS and TWIiS for Example 3: 113×143, using the Horseshoe prior.
Case: Asymmetric main & interaction effects

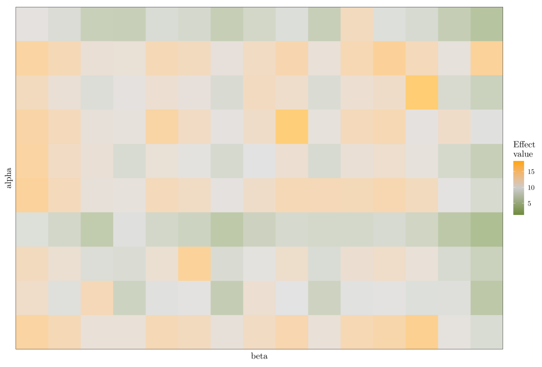
Code	Variable	ESS/sec	
		10,000 observations	80,000 observations
		(25,232 sec)	(77,324 sec)
	γs & τs	0.13	0.04
JAGS	μ	0.04	0.002
	αs	0.07	0.005
	βs	0.14	0.003
		(4657 sec)	(11333 sec)
	γs & τs	1.09	0.44
TWIiS	μ	0.24	0.015
	αs	0.44	0.04
	βs	0.71	0.024

7.4.2 Results from Rainier

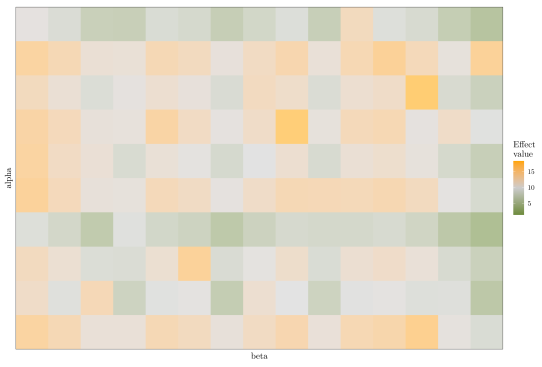
In this part we explore Rainier's performance and correctness of results for a two-way Anova model of 10 levels for the first and 15 levels for the second categorical variable. We apply variable selection to the interactions using the Horseshoe prior approach, according to the implementational details for Rainier mentioned in Section 6.4, assuming that both the main and the interaction effects are asymmetric.

Rainier needs a longer time period to converge than JAGS and TWIiS. The initial warm-up iterations were not enough to achieve convergence for one million iterations of the HMC with a thinning factor of a hundred. Therefore, an additional burn-in period led to 5,000 samples, which is the sample size used for the following analysis for inference. The posterior point estimates returned from Rainier are very similar to the ones estimated from JAGS and TWIiS, with the all group means being close to the expected true values, confirmed by the similar patterns of the equivalent heatmaps shown in Figure 7.28.

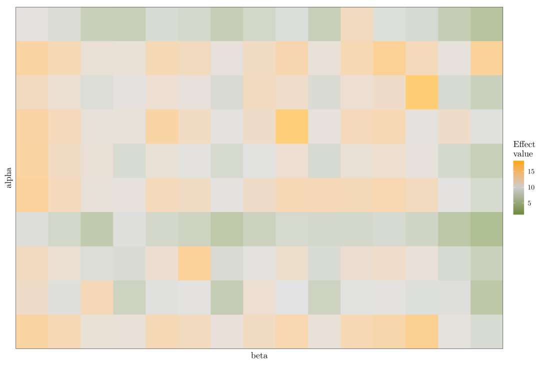
The produced diagnostic plots showed that regarding the interaction effects Rainier's



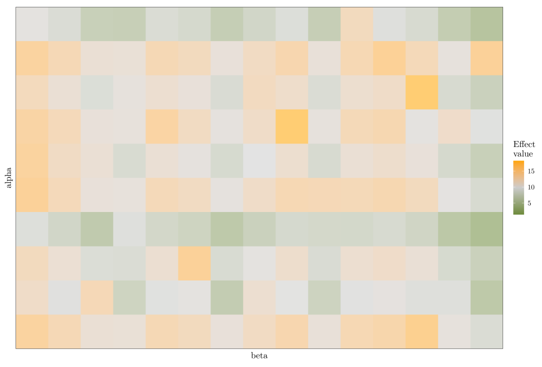
(a) JAGS



(b) TWIiS



(c) Rainier

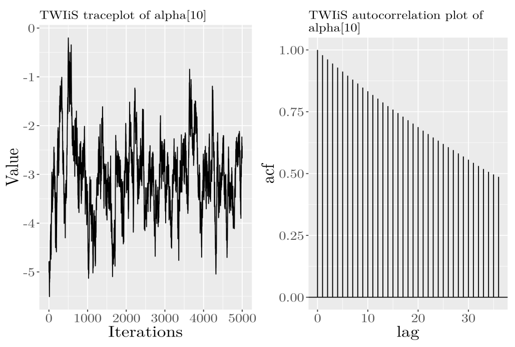


(d) True group means

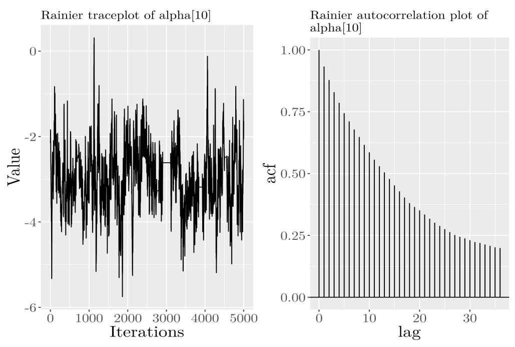
Figure 7.28: Comparison of the group means among JAGS, TWIiS Rainier and the true coefficients. The y-axis represents the levels of the first and the x-axis of the second categorical variable. *Case: Asymmetric main & interaction effects with 10×15 levels.*

mixing is poorer and the autocorrelation is higher compared to the other programs considered. This applies in particular for interaction coefficients closer to zero. Regarding the mixing of the main effects and μ there is margin for improvement for both Rainier and TWIiS as shown in Figure 7.29. JAGS presents similar results. Increasing the number of iterations improves mixing and reduces the autocorrelation between the samples. However, for this particular example, longer MCMC runs for Rainier are time demanding compared to the runtimes of JAGS and TWIiS, and therefore not further explored in this part.

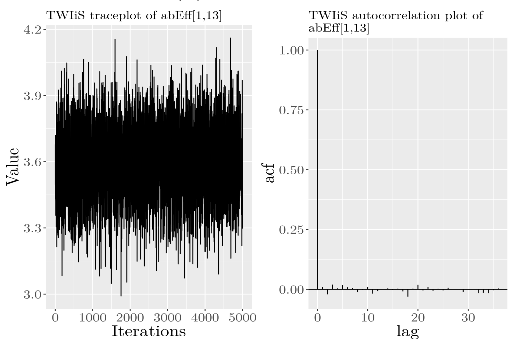
Regarding the efficiency measurements, the first 5,000 samples after having achieved convergence were used for each program. Table 7.15 summarises the runtimes and the ESS/sec for the three implementations. Rainier presents the highest ESS values for the main effects and μ, however there is a significant difference in the equivalent values for the interaction effects. It is likely that HMC draws more efficient samples for the main effects in earlier stages of the chain, as opposed to Gibbs sampler that appears to be more efficient for the interactions, as we saw in earlier cases explored. Rainier's runtime is higher and



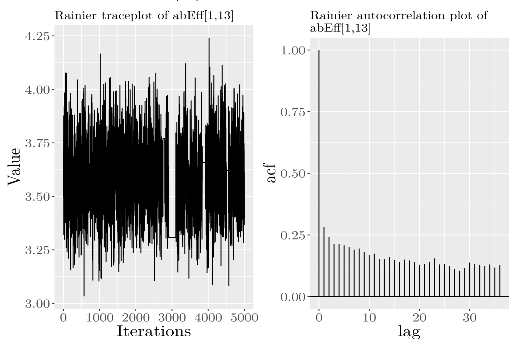
(a) TWIiS α_{10}



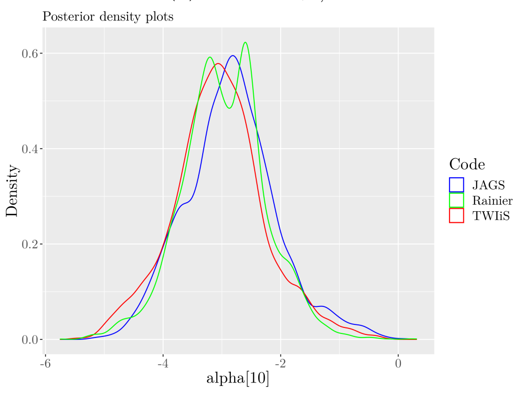
(b) Rainier α_{10}



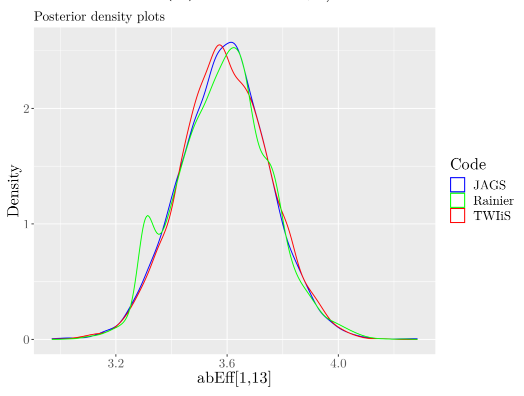
(c) TWIiS $\gamma_{1,13}$



(d) Rainier $\gamma_{1,13}$



(e) Posterior densities α_{10}



(f) Posterior densities $\gamma_{1,13}$

Figure 7.29: Diagnostic plots for α_{10} (top row) and γ_{1-13} (middle row), and posterior density plots comparison (bottom row). *Case: Asymmetric main & interaction effects with 10×15 levels.*

the ESS/sec is significantly lower than the two alternatives tested. Hence, Rainier appears to be less efficient for the particular model and example that we study.

Table 7.15: Efficiency comparison between JAGS, TWIiS and Rainier. Variable selection using Horseshoe prior. *Case: Asymmetric main & interaction effects with 10×15 levels.*

Variable	ESS			ESS/sec		
	JAGS	TWIiS	Rainier	JAGS	TWIiS	Rainier
				(1305 sec)	(155 sec)	(15659 sec)
γs & τs	3,945	4,252	143	3.02	27.43	0.09
μ	38	31	133	0.03	0.20	0.009
αs	64	44	125	0.05	0.29	0.008
βs	100	110	128	0.08	0.71	0.008

The code for developing variable selection with Rainier using Horseshoe priors for a two-way Anova model is available at the GitHub repository TWI_HS_Rainier [1].

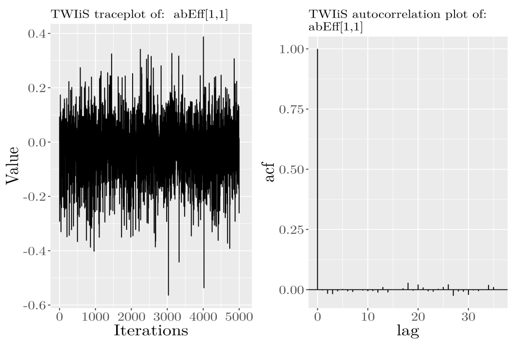
7.4.3 Results from Stan

Stan is explored as alternative software that uses HMC and the Horseshoe prior to implement variable selection on the hierarchical two-way Anova problem that we examine. The synthetic dataset used for this part is the same as for Rainier, with 15 levels for the first and 20 levels for the second categorical variables. For this model we adopt the regularised Horseshoe approach with tuned step-size and tree-depth for the HMC as discussed in Section 6.3.

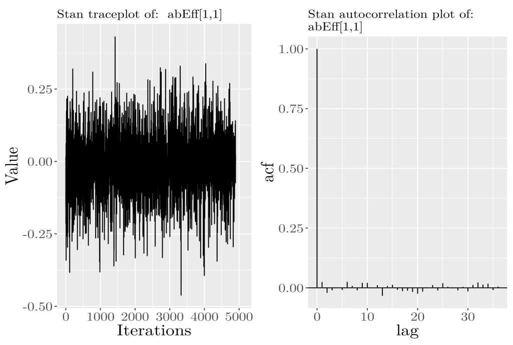
Stan, particularly after having tuned the necessary parameters applying the optimisation techniques mentioned in Section 6.3, appears to be time demanding. Consequently, the results presented in this part for inference regarding Stan derive from 100,000 iterations with a thinning factor of 10, where the program was running for 76 hours. Potential further optimisation techniques were not explored in the scope of this project. Despite the long runtime the algorithm showed convergence and presented no autocorrelation between the samples. On the other hand, for this number of iterations TWIiS and JAGS did not show convergence for some variables. Therefore, these two programs were set to run for one million iterations with a thinning factor of a hundred which corrected the previous diagnostics. This run was also used for calculating their efficiency metrics. The corresponding runtime for JAGS was 43 minutes and for TWIiS only five. Indicatively, Figure 7.30 illustrates the traceplots and autocorrelation plots of two interaction effects, as well as their corresponding posterior densities. We can observe that the estimated posterior densities from all three programs overlap, indicating their accordance.

Regarding the ESS calculation of Stan, another run of 10,000 iterations was considered without thinning since it presented good mixing and retained autocorrelation. For all three

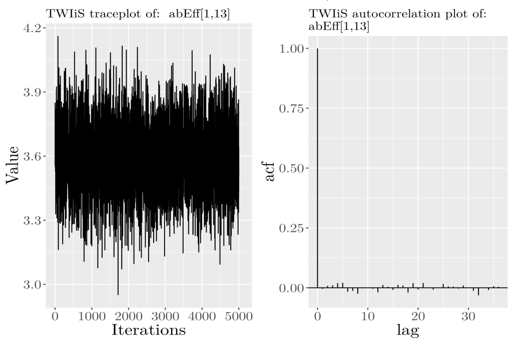
[1] TWI_HS_Rainier, `https://github.com/antoniakon/TWI_HS_Rainier`.



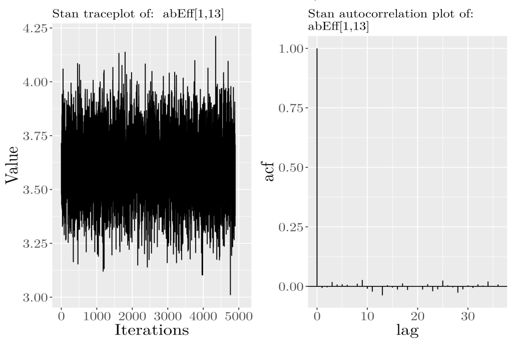
(a) TWIiS $\gamma_{1,1}$



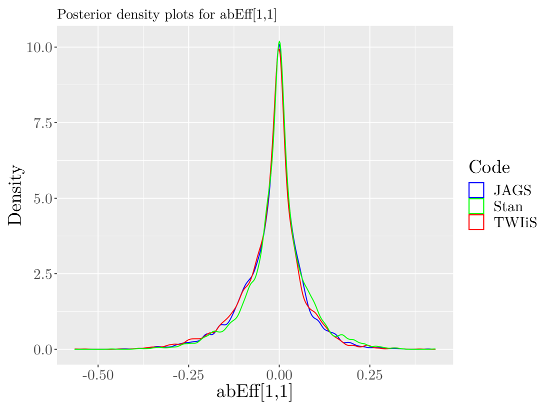
(b) Stan $\gamma_{1,1}$



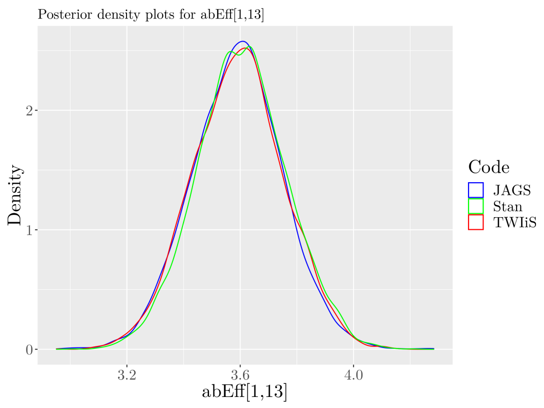
(c) TWIiS $\gamma_{1,13}$



(d) Stan $\gamma_{1,13}$



(e) Posterior densities $\gamma_{1,1}$



(f) Posterior densities $\gamma_{1,13}$

Figure 7.30: Diagnostic plots for $\gamma_{1,1} = 0$ (top row) and $\gamma_{1,13} = 3.58$ (middle row), and their posterior densities (bottom row). *Case: Asymmetric main & interaction effects with 10×15 levels.*

programs the first 5,000 samples after burn-in and convergence were used for calculating the ESS and efficiency measurements which are summarised in Table 7.16 that displays the minimum values for each category of variables. Looking at the individual categories we can see that Stan presents high ESS values for the main and low for the interaction effects as opposed to TWIiS and JAGS. This could be due to the fact that HMC might present better mixing for some parameters and Gibbs sampling for others. Comparison of the minimum ESS/sec from all three programs shows that Stan has similar performance to JAGS, with TWIiS being more efficient. However, looking at the individual categories Stan seems to have better performance for the main effects.

Table 7.16: Efficiency comparison between JAGS, TWIiS and Stan. Variable selection using Horseshoe prior. *Case: Asymmetric main & interaction effects with 10×15 levels.*

Variable	ESS			ESS/sec		
	JAGS	TWIiS	Stan	JAGS	TWIiS	Stan
				(1,305 sec)	(155 sec)	(1,836 sec)
γs & τs	3,945	4,252	41	3.02	27.43	0.02
μ	38	31	29	0.03	0.20	0.02
αs	64	44	2,522	0.05	0.29	1.37
βs	100	110	3,001	0.08	0.71	1.64

Finally, a heatmap of the pattern that the group means estimated by Stan follows, is shown in Figure 7.31. Comparing it to the heatmaps in Figure 7.28 illustrating the group means estimated from JAGS, TWIiS and Rainier, we can see that they all follow the same pattern.

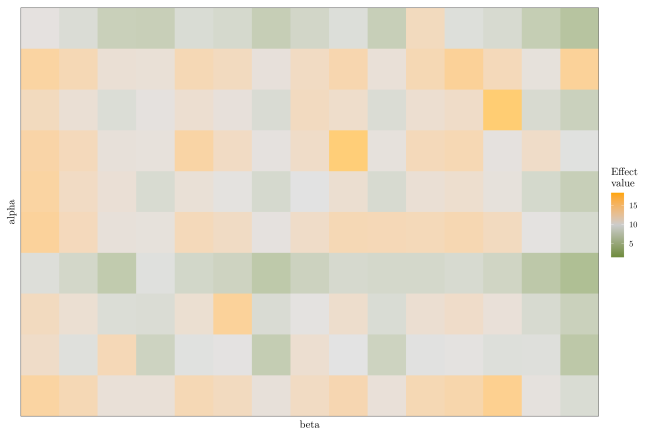


Figure 7.31: Group means heatmap for Stan. The y-axis represents the levels of the first and the x-axis of the second categorical variable. *Case: Asymmetric main & interaction effects with 10×15 levels.*

7.5 Efficiency comparison between variable selection using indicators and Horseshoe priors

So far in this chapter we have presented through various examples how variable selection can be applied on the interactions of a hierarchical two-way Anova modelling problem that we examine associated with four different cases of symmetry in the effects, using two alternative methodologies, the Kuo and Mallick approach and the Horseshoe prior. In order to assess the results and efficiency of the two variable selection methods on this particular model, it was necessary to run multiple simulations for all cases, record their runtimes and further investigate the effective samples sizes and the ESS/sec. Especially when dealing with complex hierarchical models that can be computationally demanding or when the volume of data is large, leading to increases in runtime, we are interested in both the correctness of results, and computational time and efficiency.

This section aims to summarise the performance of all methods studied in JAGS and TWIiS, facilitate performance comparisons and highlight the methodology that is more appropriate for our model in terms of speed and accuracy of results. As the real datasets of the yeast genome case study are more complex, it is worth observing how the efficiency metrics are influenced as the levels of the categorical variables change and the number of observations increase.

All the results regarding the various measurements for the example with 15 levels for the first and 20 levels for the second categorical variables are summarised in Table 7.17. There we can see that for the Gibbs sampler written in Scala, the runtimes and the ESS/sec deteriorated using the Horseshoe prior compared to the equivalent measurements coming from the implementation using an indicator for each parameter. In some cases the decrease in performance is slight and in others the program using Horseshoe priors is approximately two times slower. For JAGS, both the running time and the efficiency metric using the Horseshoe prior showed slight improvement compared to the indicator approach. The synthetic dataset used had a sample size of 10,000.

Nevertheless, when looking at the results for the second example in Table 7.18 concerning the synthetic dataset with 50 levels for the first and 60 levels for the second categorical variables for the same number of observations, we can see that the improvement observed for JAGS in the smaller example is no longer valid.

After examining the two previously mentioned cases, an example with number of levels equal to the yeast genome case study dataset was explored, in order to test the performance of the two implementations at a larger scale. This model has 113 levels for the first and 143 levels for the second categorical variable, with a sample size of 10,000 observations and assumes that both the main and the interaction effects are asymmetric. The results for this case are illustrated in Table 7.19, at the top row of each case. There, we can see that the implementation of variable selection with indicators preforms better than the one using Horseshoe priors. This difference is more significant for JAGS where the first implementation appears to be 3.5 times faster than the second, whereas for TWIiS, this ratio is 1.6.

In order to have a more complete understanding of the pattern in the efficiency of the two methodologies, and be more certain about which implementation appears to work better for our modelling process, we also considered a last synthetic example that also has 113×143 levels, but 80,000 observations. This last example intends to reflect more the actual structure of the yeast genome dataset and model, and its results are presented in Table 7.19 in the bottom row of each case. Again, for both cases variable selection using indicators appears to be more performant than the Horseshoe prior, even though their difference for TWIiS is not highly significant. However, the method that uses variable selection indicators has faster computational time compared to the Horseshoe alternative, meaning that longer runs often required for inference are easier to implement with the first technique.

Another comparison that would be interesting is to see how the efficiency differences are formed for the case that is common in all examples, the model where both the main and the interaction effects are considered to be asymmetric. For the smallest example, TWIiS is approximately nine times more efficient than JAGS for the indicator approach, a ratio that drops to five for the Horseshoe prior implementation. The corresponding values for the second and third examples are approximately three times for the indicator approach and they remain similar for the Horseshoe prior. These observations lead to the conclusion that the increase in the complexity of the model reduced the difference in the performance between the two implementations for the variable selection indicator, but did not affect the Horseshoe prior approach in the same way. However, this difference increases again as the sample size increases in the last example showing that TWIiS probably processes more data in a faster way, since the ESS is similar in both programs with the different sample sizes examined.

Regarding the overall performance of JAGS and TWIiS, looking at the case where the main and interaction effects are asymmetric for all examples with 10,000 observations shows that the ESS/sec metrics for JAGS are more consistent across the examples for the case of the Horseshoe prior and present greater variation when the variable selection indicators are used. TWIiS shows greater variation in the efficiency metric across both implementations. Both performance metrics change significantly when the sample size increases.

To sum up, efficiency comparison across all examples explored showed that for both JAGS and TWIiS, variable selection using the indicator approach appears to have better performance than the equivalent implementation that combines a Gibbs sampler and

Horseshoe priors. Furthermore, TWIiS is generally more efficient than JAGS. The extent to which TWIiS and the variable selection using indicators method are more performant than their counterparts depends highly on the model case in terms of symmetry in the effects, complexity and sample size.

As a remark about the efficiency difference between the implementations in TWIiS and JAGS, detailed exploration of the JAGS source code is beyond the scope of this project. However, regarding TWIiS, using long established collections and operations from the JVM leads to performant code that lets programmers focus on the high-level of implementing the algorithm, while leaving the low-level operations such as memory management to the JVM. Using Scala and functional code has an additional advantage of guarding against bad object-oriented practices, such as keeping states in mutable variables in memory longer than actually needed and thus wasting computer resources. Additionally, functional code and immutable collections allow for safe leverage of computer resources by using parallelism that depending on the problem leads to further speed-up as we saw in Section 6.2.4. Intentionally, we avoided using the parallel code of TWIiS, to provide a fairer comparison among the examined implementations.

Table 7.17: Efficiency comparison between JAGS and TWIiS for variable selection using indicators or the Horseshoe prior. Example 1: 15×20 levels.

Case	VSI				Horseshoe			
	ESS		ESS/sec		ESS		ESS/sec	
	JAGS	TWIiS	JAGS	TWIiS	JAGS	TWIiS	JAGS	TWIiS
Both asym.	μ: 66	μ: 69	0.03 (2,193s)	0.29 (242s)	μ: 83	μ: 97	0.05 (1,699s)	0.25 (382s)
Sym. main	Z: 85	Z: 92	0.04 (2,018s)	0.4 (220s)	μ: 83	μ: 82	0.05 (1,815s)	0.22 (378s)
Sym. inters	μ: 106	μ: 93	0.05 (2,243s)	0.31 (297s)	α: 37	μ: 63	0.03 (1,559s)	0.13 (481s)
Both sym.	Z: 83	Z: 86	0.04 (1,928s)	0.31 (278s)	Z: 93	Z: 72	0.06 (1,542s)	0.21 (342s)

Table 7.18: Efficiency comparison between JAGS and TWIiSfor variable selection using indicators or the Horseshoe prior. Example 2: 50×60 levels.

Case	VSI				Horseshoe			
	ESS		ESS/sec		ESS		ESS/sec	
	JAGS	TWIiS	JAGS	TWIiS	JAGS	TWIiS	JAGS	TWIiS
Both asym.	μ: 339	μ: 296	0.1 (3,436s)	0.32 (912s)	μ: 319	μ: 321	0.04 (7,254s)	0.18 (1,835s)
Sym. main	Z: 144	Z: 136	0.03 (4,421s)	0.09 (1,583s)	μ: 147	μ: 153	0.02 (8,564s)	0.07 (2,075s)
Sym. inters	μ: 301	μ: 353	0.1 (2,993s)	0.37 (958s)	μ: 370	μ: 394	0.07 (4,979s)	0.22 (1,773s)
Both sym.	Z: 163	Z: 140	0.05 (3,643s)	0.12 (1,105s)	μ: 145	μ: 152	0.03 (4,972s)	0.08 (2,016s)

Table 7.19: Efficiency comparison between JAGS and TWIiS for variable selection using indicators or the Horseshoe prior. Example 3: 113×143 levels.
Case: asymmetric main and interaction effects.

Method	Case	ESS		ESS/sec	
		JAGS	TWIiS	JAGS	TWIiS
VSI	10k obs	μ: 1,173	μ: 1,082	0.14 (8,543s)	0.38 (2,912s)
	80k obs	μ: 139	μ: 139	0.004 (51,599s)	0.02 (7,326s)
Horseshoe	10k obs	μ: 1,103	μ: 1,105	0.04 (25,232s)	0.24 (4,657s)
	80k obs	μ: 128	μ: 174	0.002 (77,324 s)	0.015 (11,333 s)

Chapter 8

Yeast robotic genetic studies

Bayesian hierarchical modelling has a broad range of applications. As a case study in this project we explore a dataset, and the corresponding model, that comes from the field of genetics. The data on which we apply the methods developed derive from the Institute for Cell and Molecular Biosciences in Newcastle University (ICaMB) and are related to the yeast genome and telomere biology.

Telomeres are regions at the ends of the chromosomes that protect the genetic information. Chromosome replication and various factors, such as ageing, cause telomere shortening. Exploring telomere capping defects and potential interactions with specific genes on model organisms, such as yeast, helps scientists understand the mechanism behind enhancement or suppression of the phenomenon.

The aim of this chapter is to establish the biological background and describe the genetic experiment and process that leads to the resulting dataset. We will also present the modelling specifications of the case study that is subject to analysis.

8.1 Introduction to genetics – Important terms

In this section we will provide an introduction to some fundamental terms in genetics that will be used in the following sections and are helpful to understand the nature of the experiment.

8.1.1 DNA and RNA

Deoxyribonucleic acid (DNA) is a molecule found in the cells of an organism and constitutes the hereditary material that contains the genetic information and instructions for its growth and functioning. The DNA has two helical chains coiled around each other and is made up of monomers called nucleotides. Nucleotides have three components; a 5-carbon sugar, a phosphate group and a nitrogen base. There are four nitrogen bases, cytosine (C), guanine (G), adenine (A) and thymine (T), and the hydrogen bonds that connect them according to the base pairing rules (A with T and C with G) allow for the two strands to be held together. Each strand is shaped due to the chemical bonds developed between the sugar group of the one nucleotide and the phosphate group of the next. An organism's total genetic information is called *genome*. Another molecule essential in various biological roles is Ribonucleic acid (RNA), which unlike DNA has a single strand. An additional difference from DNA is that instead of the nitrogen base thymine (T), RNA has uracil (U).

Both DNA and RNA have an end-to-end chemical orientation. They are synthesised in the $5'$ to $3'$ direction, which is also assumed to be the default unless it is otherwise defined. The $5'$-end of a strand is named after the fifth carbon in the sugar-ring of the deoxyribose or ribose and the $3'$-end refers to the hydroxyl group of the third carbon in the sugar-ring that is the terminating chemical element.

For the double-stranded DNA helix the strands are antiparallel to each other, so that the nucleotides of the one strand are complemented by the nucleotides of the second strand as shown below (Hartl and Jones, 1998):

$5'$...CTCAATGTTCGA...$3'$ (also called the Watson strand)
$3'$...GAGTTACAAGCT...$5'$ (also called the Crick strand)

8.1.2 Chromosomes

In the nucleus of cells, the DNA is packed around proteins which are called histones and is packaged into structures that are called chromosomes. Humans have 23 pairs of chromosomes. Two of the 46 chromosomes determine the sex of the person so they are called sex chromosomes (X and Y), while the rest are called autosomes. Chromosomes are not visible under a microscope in every phase of the cell's life. This happens only during a specific phase (metaphase) of the cell division (mitosis) and a picture of a person's chromosomes is called a karyotype. The karyotype of a human is presented in the karyogram displayed in Figure 8.1. The final pair of chromosomes indicates the sex of the individual. In this figure the one chromosome (Y) is smaller than the other (X) which indicates that the DNA comes from a male. Finally, during mitosis when the chromosome replicates and two identical copies of the DNA are formed (sister chromatids), both copies are joined together by a specialised DNA sequence known as "centromere" (Hartl and Jones, 1998). Centromeres are used as a reference point for the identification of the position of various DNA sequences (genes) on the chromosome. Figure 8.2 illustrates the relations among some of these molecules.

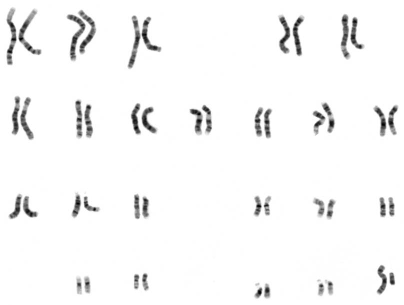


Figure 8.1: Karyogram of human male (National Human Genome Research Institute, 2020)

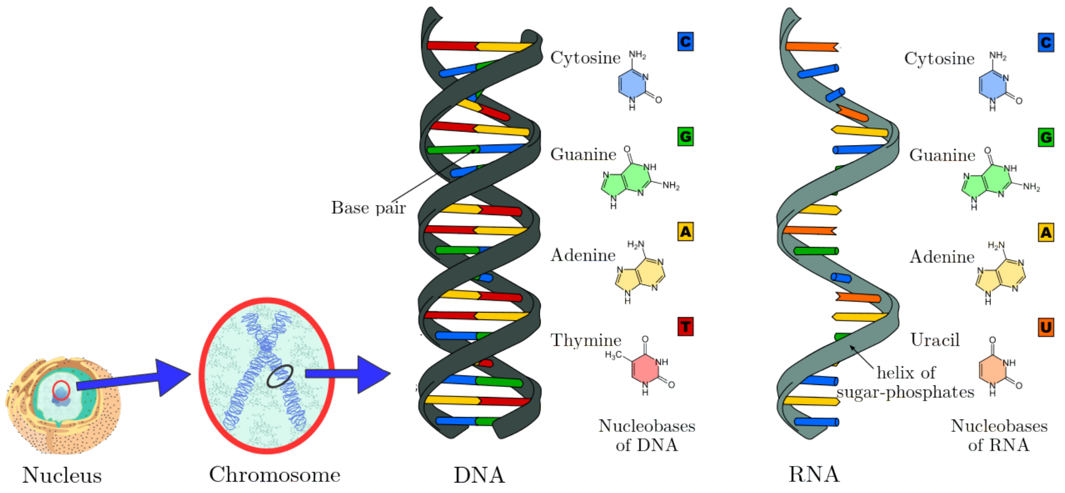


Figure 8.2: Chromosomes - DNA - RNA. [*Image adapted from Wikimedia Commons (2012)*]

8.1.3 Genes and proteins

Genes are small sections of the chain of DNA and vary in size. Chromosomes contain many genes, for example chromosome 1, which is the largest in humans, contains about 8,000 genes. Each gene is situated at a specific location on the chromosome, called the locus, in two copies inherited from each parent. A variant form of a gene is called allele. Each of the two copies of a gene can be different, for example people with blood type AB have one allele A and one B. Most genes are protein-coding, meaning that they provide the genetic information to define the building blocks, called amino acids, for the synthesis of proteins after gene expression. Gene expression involves the process of DNA's transcription to RNA, which then specifies the order by which amino acids will be linked together by the ribosomes during translation and synthesise proteins. A triplet of adjacent nucleotides is called a codon and corresponds to a specific amino acid or start and stop signals during protein synthesis. Proteins are macromolecules essential to all living organisms. They are necessary for the structure and function of the body's tissues and organs (MITx, 2013).

According to the Human Genome Project humans have between 20,000 and 25,000 protein coding-genes (International Human Genome Sequencing Consortium, 2004).

8.1.4 Genotype, phenotype and mutations

By the term genotype, scientists refer to the set of genes that an organism has, whereas phenotype refers to its observable characteristics. Phenotype is affected by the genotype and by environmental factors. In order to describe the allele that encodes the phenotype that is the most common in nature we use the term wild-type allele (MITx, 2013).

Permanent changes in the genetic sequence are called mutations. Mutations can be recessive or dominant. In order for the mutant phenotype to be observed, in the first case both alleles have to be mutant (homozygous individual) whereas in the second, the individual can carry one mutant and one normal allele (heterozygous) (Lodish *et al.*, 2000). In some experiments scientists mutate genes of interest and the organism is then called mutant. Mutating alleles in two genes of interest renders the organism "double mutant".

8.1.5 Open Reading Frame (ORF)

During protein synthesis, the nucleotides are read in the $5' \rightarrow 3'$ direction and are grouped into consecutive and non overlapping triplets. There are three different ways that a sequence can be read, depending on which is considered the first nucleotide of the reading frame. Consequently there are three different reading frames as shown in the example bellow:

Sequence: 5′...CTCAATGTTCGACGA...3′
Reading Frame 1: 5′...CTC-AAT-GTT-CGA-CGA...3′
Reading Frame 2: 5′...C-TCA-ATG-TTC-GAC-GA...3′
Reading Frame 3: 5′...CT-CAA-TGT-TCG-ACG-A...3′

Some triplets can be translated to amino acids and we mentioned in Section 8.1.3 that they are then called codons. There are also specific codons in the DNA that act as signals that can start or stop the translation process. The start (ATG) or stop codons (TAA, TAG or TGA) define the Open Reading Frames, which are the sequences of bases between them. These sequences are actually DNA sequences that have the potential to be transcribed into RNA and translated into proteins. Their identification is very important in tracing genes (Hartl and Jones, 1998).

8.1.6 Telomeres

Telomeres (Figure 8.3) are specific DNA-protein structures found at the ends of the chromosomes of eukaryotic organisms. Their length in humans varies according to age but they are around 10 kilobases long. Telomeric DNA consists of hundreds of repetitions of a short sequence of bases, which for humans is *TTAGGG*. Telomere capping protects the chromosome from degradation every time it replicates during cell division and consequently plays a vital role in the preservation of the genome. However, during each DNA replication hundreds of bases are lost, resulting in telomere shortening. When telomere length reaches a critical limit the cell can no longer replicate, causing cell malfunction (Aubert and Lansdorp, 2008) or even death (apoptosis). In most cells an enzyme called telomerase acts against telomere shortening by synthesising telomeric DNA. Nevertheless, in some cells, e.g. somatic cells, the activity of the enzyme takes place at lower levels therefore these cells undergo a faster shortening.

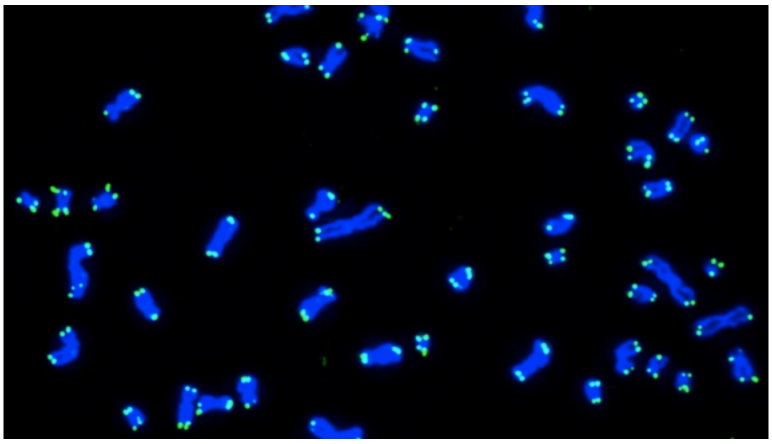


Figure 8.3: Fluorescence-stained chromosomes (blue) and telomeres (green) under a microscope during the metaphase of the cell cycle (Adam *et al.*, 2019).

The significance of telomeres in an organism's function and maintenance renders their study an important topic in the field of genetics. A lot of research has been conducted regarding how telomere shortening is related to ageing and cancer (Maciejowski and de-Lange, 2017). There is evidence that telomere shortening is linked to ageing. Short telomere length for specific age groups can also be associated with increase in age-related diseases (Herrmann *et al.*, 2018). In addition, factors such as obesity, stress, dietary habits, as well as environmental factors seem to affect telomere shortening (Shammas, 2011).

8.1.7 Saccharomyces cerevisiae (budding yeast)

Saccharomyces cerevisiae (Figure 8.4) is commonly known as baker's yeast or budding yeast and is a single-celled eukaryotic organism. It is widely used in scientific research because its genome has been sequenced and it constitutes an organism which is easy to grow and genetically manipulate in vitro. It has a cellular organisation that is similar to that of higher eukaryotes like humans, so its genes' functions help scientists understand more complex organisms.

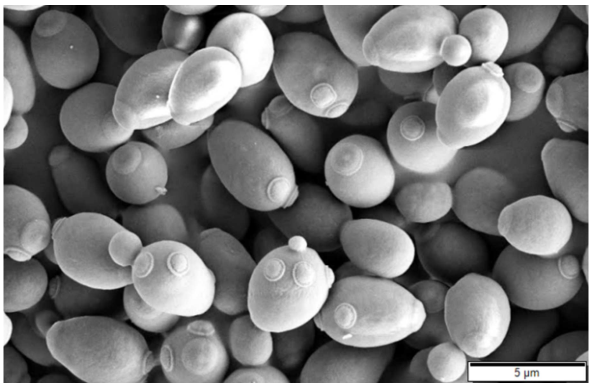


Figure 8.4: *Saccharomyces cerevisiae* under scanning electron microscope (DasMurtey and Ramasamy, 2016).

S. cerevisiae is widely studied and this led to the Saccharomyces Genome Database (2020)(SGD) which constitutes a community resource and scientific database for budding yeast. The SGD provides information about the yeast genome, interactions between genes and pathways, datasets and experiments as well as papers and various resources. According to the SGD there is a standard and consistent nomenclature for a yeast locus, a Systematic ORF Name and a Standard Gene Name. The Systematic ORF name consists of three uppercase letters followed by a 3-digit number and a letter.

As an example we will use the open reading frame YFL039C:

- Y stands for 'Yeast'.
- The second letter is in the range A - P. It denotes the chromosome upon which the ORF resides. *S. cerevisiae* has 16 chromosomes and 'A' is chromosome I, up to 'P' for chromosome XVI.
- L in this example stands for 'Left' and indicates that the ORF is on the left arm of the chromosome. Similarly, R denotes that it is on the right arm.
- The next 3-digit number corresponds to the sequential order of the ORF on the arm (with direction from the centromere to the telomere).
- W or C denotes whether the ORF is encoded on the 'Watson' or 'Crick' strand.

So, the ORF YFL039C is the 39th ORF to the left of the centromere on chromosome VI, and is on the Crick strand.

Sometimes an ORF is also given a standard gene name that consists of three letters followed by a number. Dominant alleles, usually wild-type, are denoted by uppercase letters and recessive or mutant alleles by lowercase. The three letters denote a description of the phenotype or a known gene function. For example, *CDC13* is a gene name for the gene that encodes for the *Cell Division Control* protein 13, Cdc13, with the ORF name YDL220C. Respectively *cdc13-1* refers to a temperature sensitive mutant that involves a telomere capping defect that inhibits the cell-cycle progression. This mutant is of interest as *CDC13* is essential and cannot be studied using deletion.

8.1.8 Epistasis

As mentioned in Section 8.1.6, telomeres are specialised structures at the end of the chromosomes that protect the genome every time the cell divides. Various non-essential proteins contribute to telomere capping and when there is a mutation in the genes coding for these proteins, telomeric DNA and the cell cycle do not function properly. For example, Cdc13 is a constituent of a protein complex that takes part in telomerase recruitment. When the *cdc13-1* mutant allele is present there can be an acute inactivation of the protein Cdc13 that can even cause cell death. The reason for this is that *cdc13-1* is a temperature sensitive mutant and at specific temperatures it causes ssDNA (single stranded DNA) generation and accumulation at telomeres which disrupts their function and pauses the cell cycles. For example, there is evidence that deleting the gene *EXO1* that is responsible for the production of Exo1 nuclease that contributes to ssDNA production when Cdc13 is defective, can suppress the poor growth of the cells caused by the presence of the mutant allele. This is a phenomenon of genetic interaction between those two genes and it is called epistasis. In this example the poor growth is suppressed, however there are other examples where the presence of some genes can enhance the poor fitness caused by already defective telomeres (Addinall *et al.*, 2011).

The length of the telomeres is a complex characteristic determined by a large number of genes. According to Ungar *et al.* (2009) and his reference to previous genome-wide studies on yeast, there are 272 non-essential genes that can cause telomere shortening or lengthening when they are mutated.

8.2 Quantitative Fitness Analysis

Quantitative Fitness Analysis (QFA) is an experimental and computational workflow aiming to compare fitnesses of microbial cultures that, after being subjected to applied treatments, are spotted onto agar plates which are then incubated and regularly photographed (Banks *et al.*, 2012). Telomeres degrade and shorten over time. In addition, genes associated with proteins that contribute to telomere capping can be mutated leading to capping defects. The aim of the experiment examined in this study is to better understand telomere biology in budding yeast by identifying genetic interactions between non-essential genes, which are genes that are not essential for the viability of the organism, and genes that are related to telomere capping. For the conduction of the experiment, scientists in the Institute of Cell and Molecular Biosciences use Synthetic Genetic Array (SGA), a methodology which is analytically described in the following part.

8.2.1 Synthetic Genetic Array and gene deletion

Synthetic Genetic Array is a methodology developed by Tong and Boone (2006) for identifying genetic interactions between genes. It is based on the construction of double mutants using gene deletion, where a query mutation is crossed to viable gene deletion mutants. Scientists look at fitness[1] defects in various screens[2] and come to conclusions about potential interactions between the genes involved.

In SGA gene deletion actually refers to gene substitution with cassettes. Cassettes are DNA molecules that act as mobile elements transferring foreign genetic material into another cell. In gene deletion, cassettes often replace a specific gene with a gene that encodes for resistance to a certain antibiotic. The position of the gene to be replaced is determined by a 'genetic barcode' carried by the cassette. Suppose that the cassette natMX, providing antibiotic resistance to nourseothricin, replaces in various yeast strains the ORF with the gene name GeneX. Then, only the strains with the antibiotic resistance will be able to survive when they get transferred and allowed to culture in agar plates containing the specific antibiotic, in this case nourseothricin. A second deletion can also take place on the same strain by using another antibiotic resistance cassette, such as the kanMX that provides resistance to kanamycin (Tong and Boone, 2006). A visual representation of the gene deletion logic is shown in Figure 8.5.

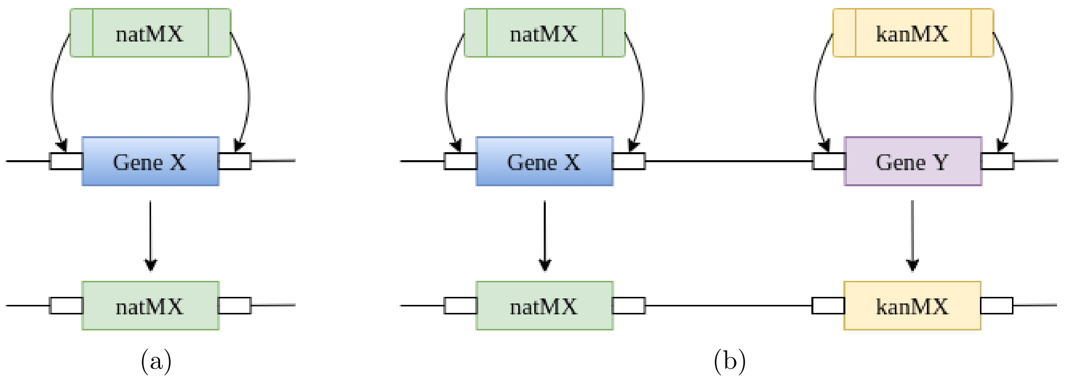


Figure 8.5: Gene deletion by displacement (a) Gene X from the cassette natMX, gene xΔ::natMX (b) additional Gene Y from the cassette kanMX, gene yΔ::kanMX (Tong and Boone, 2006).

[1]Quantitative representation of organisms' survival and reproduction in a given environment; acts as a measurement of growth (Heydari *et al.*, 2016).

[2]Technique used for identification and selection of individuals with specific traits coming from a population subjected to mutations (Heydari *et al.*, 2016).

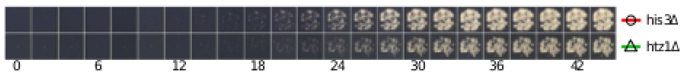
8.2.2 QFA – Overview of the experiment and methodology

QFA aims specifically to identify epistasis between genes that affect telomere capping causing telomere defects and certain non-essential genes, as in the example presented in Section 8.1.8 where *EXO1* was deleted. In budding yeast there is also another non-essential protein, Yku70, lack of which, due to deletion of gene YKU70 (yku70*Δ*), can cause telomere shortening and temperature sensitivity, as with the case of the *cdc13-1* mutant allele.

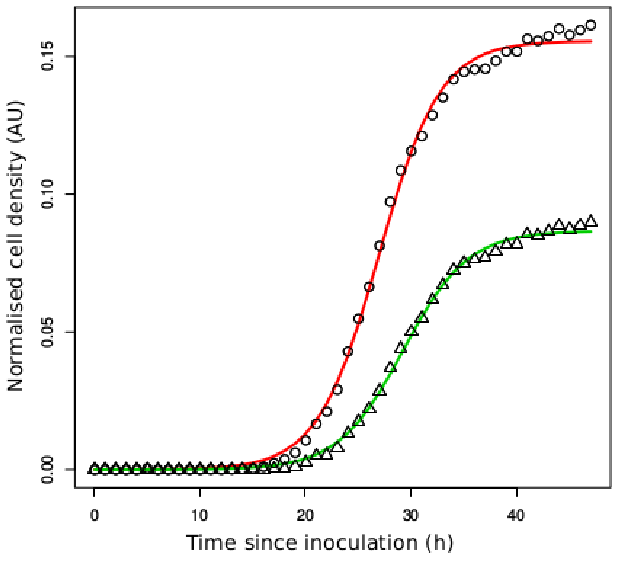
To clarify some terms used in the description of the experiment we can summarise the following:

- *Knockout library*: ∼4200 viable gene deletions, all related to non-essential genes. We denote an arbitrary gene deletion as *yfgΔ* (Your Favourite Gene Deletion), or sometimes as *orfΔ* when we refer to a deletion of a specific ORF.
- *Query strain*: Yeast strain containing one of the *cdc13-1* and *yku70Δ* mutations that cause temperature sensitive telomere defects.
- *Control strain*: Yeast strain containing the neutral *ura3Δ* mutation that has no effect on the fitness under the experimental conditions. It does not interact with any *yfgΔ* and so it is also described as the "wild-type strain" in some contexts.
- *Double mutants*: Yeast strains that contain both *cdc13-1* or *yku70Δ* mutation and a non-essential gene deletion (*yfgΔ*).

For QFA, scientists use robotic facilities and Synthetic Genetic Array (SGA, Section 8.2.1) methodology to combine the systematic gene deletion collection (knockout library) with *cdc13-1* and *yku70Δ* mutations. Then, the double mutants are spotted onto agar plates with ∼100 cells placed in each of the 384 spots of each plate. Next, the plates are incubated under different temperatures, due to the temperature sensitivity of the mutants, and the yeast colonies start to form (Figure 8.6a). The growth of the colonies under the different treatment conditions is estimated by the use of time course photography. Multiple images of the plates are taken at close time intervals to record the progress of the cultured yeast strains. These photographs are analysed through image processing and after obtaining the corresponding growth curves (Figure 8.6b), a logistic growth model is fitted to estimate the quantitative parameters that describe the fitness (Addinall *et al.*, 2011). One of the resulting estimated parameters is the maximum doubling rate (MDR) measured in population doubling/day which shows the rate at which the strains divide. Furthermore, the maximum doubling potential (MDP, population doublings) is estimated to capture the number of divisions before the culture is observed to undergo saturation. Their product



(a)



(b)

Figure 8.6: QFA image data and growth curves: (a) time lapse images for two genetically modified *S. cerevisiae* (b) corresponding cell density estimated through time (Heydari *et al.*, 2016).

accounting for both attributes is also sometimes used as a measurement of fitness (Heydari *et al.*, 2016).

More than four million images are captured, processed and analysed in total during the experiment with the use of an image analysis tool called Colonyzer. It constitutes an open-source collection of algorithms aiming to automate quantification of important parameters of cultures grown on solid agar (Lawless *et al.*, 2010).

8.2.3 Quantifying genetic interactions

The objective of the experiment is to identify genetic interactions between the "deleted" genes and the mutant genes examined and quantify the strength of the existing interactions. According to Mani *et al.* (2008) and Addinall *et al.* (2011), based on Fisher's multiplicative model of genetic independence, the relative fitness of the double mutants $F_{xyz\,yfg\Delta}$ should be equal to the product of the corresponding single mutant relative fitness values. Supposing that the fitness of the control strain is $F_{ura3\Delta}$, of the query strain is $F_{cdc13\text{-}1}$ and that the query strain *cdc13-1* is crossed with *orfΔ*, the expected fitness for the double mutant should be:

$$F_{cdc13-1\,orf\Delta} = \frac{F_{cdc13-1}}{F_{ura3\Delta}} \times F_{ura3\Delta\,orf\Delta},\ \forall orf\Delta. \tag{8.1}$$

In Equation 8.1, the ratio of fitness of the query over the control strain is a constant, so we expect a linear relationship between the fitness of the query strain with the orf*Δ* and the control strain with the same non-essential gene deletion (Heydari *et al.*, 2016). A deviation between the observed and the expected fitness calculated from Equation 8.1 would indicate that there is a genetic interaction between the query *cdc13-1* mutation and the deleted gene. The genetic interaction strength (GIS) is determined by the size of this deviation.

Alternative approaches to the multiplicative model chosen by Addinall (Addinall *et al.*, 2011) could be the additive model on the log-scale, the product or the minimum. Results of the identification of genetic interactions are described by Mani *et al.* (2008) who applied the various methods on two case studies.

8.3 miniQFA

MiniQFA is a research project based on the QFA described in the previous section. The objective is to find genetic interactions between telomeres that show capping defects and simultaneous deletion of two non-essential genes, instead of one that was deleted for the case of the QFA. The miniQFA experiment constitutes our main case study.

8.3.1 miniQFA overview

Proteins play an essential role in an organism's function and existence. Sometimes they interact with each other to perform a specific function. A collection of such proteins is called a protein complex (Levy *et al.*, 2006). If one or more of the genes that encode for the proteins in a complex are mutated, the biological pathway will be hindered or inhibited and this will lead to a potential malfunction of the cell.

Research conducted by Ungar *et al.* (2009) uncovered various essential genes that code for proteins, parts of protein complexes, mutations of which can exhibit a 'short' or 'long' telomere phenotype. These complexes are related to functions such as RNA processing and transcription, degradation of unneeded or damaged proteins and various others. An example of a complex of interest is the MRX (Mre11-Rad50-Xrs2) complex that is an important factor in the cells' response to double-strand breaks (DSB). A lot of research has been done in to how it affects telomeric DNA ends and how it regulates telomerase action, since the enzyme Tel1 (Serine/theorine protein kinase) is activated by it, supports its function and affects telomere length by promoting telomerase recruitment (Gobbini *et al.*, 2016).

MiniQFA aims to find genetic interactions between telomeres that show capping defects, and simultaneous deletion of two non-essential genes. This experiment is very important for the identification of groups of genes that act together in telomere length maintenance in *S. cerevisiae* and the recognition of equivalent genes (homologues) in other organisms such as humans.

In the miniQFA experiment scientists would like to delete all pairwise combinations from the knockout library, which includes 4294 non-essential genes. In the QFA experiment, 4135 out of those non-essential genes where actually used, and a genome-wide analysis for the miniQFA would result in $4135^2 = 17,098,225$ different pairwise combinations to check. Due to the nature of the experiment that demands mutating and cultivating yeast strains in different lab conditions, as well as observing them with regularly taken images, examining all these combinations would not be feasible even with the use of the robotic facilities designed for high throughput experiments. Consequently, a smaller collection of genes were selected from the knockout library, which led to naming the experiment *miniQFA* (Newman, 2017).

Previous experiments highlighted certain genes that affected telomere length, or were found to interact with mutants causing telomere defects or have homologues in human cells. Those genes along with some known neutral mutations, such as *his3Δ*, were chosen to compose a 154 *orfΔ* knockout library. The number 154 is not randomly selected. It is the number of spots of a 384-spot agar plate with a 24×16 grid from which edge spots are discarded as colonies that grow there have a growth advantage since there is no competition for nutrients. There are 308 spots remaining, so it was decided that 154 genes would be considered in the knockout library, leading to $154^2 = 23,716$ different pairwise combinations of deletions. On each plate either *cdc13-1* mutant or wild-type yeast strains are used. The query screen, which is the first gene deletion, is the same across all spots and there are two replicates of each of the 154 genes, that are used for the second deletion, on each plate. In addition, there are multiple plates incubated for each experiment.

An example of a 384-spot agar plate with the incubated colonies is presented in Figure 8.7.

8.3.2 Notation and multiplicative model

The notation used in the miniQFA experiment is consistent with the one followed for the QFA. A conditionally defective gene, such as *cdc13-1* which is defective at high temperatures, is crossed with the deletion of two other genes from the knockout library, for example *yfg1Δ* and *yfg2Δ*, and the triple mutant is denoted as *cdc13-1 yfg1Δ yfg2Δ*. In cases where the same ORF is targeted twice for deletion, the yeast strain is called double-l-mutant. This situation leads to cell death due to the nature of the process of applying the selected mutations presented in Section 8.1.8. The two consecutive cassettes providing

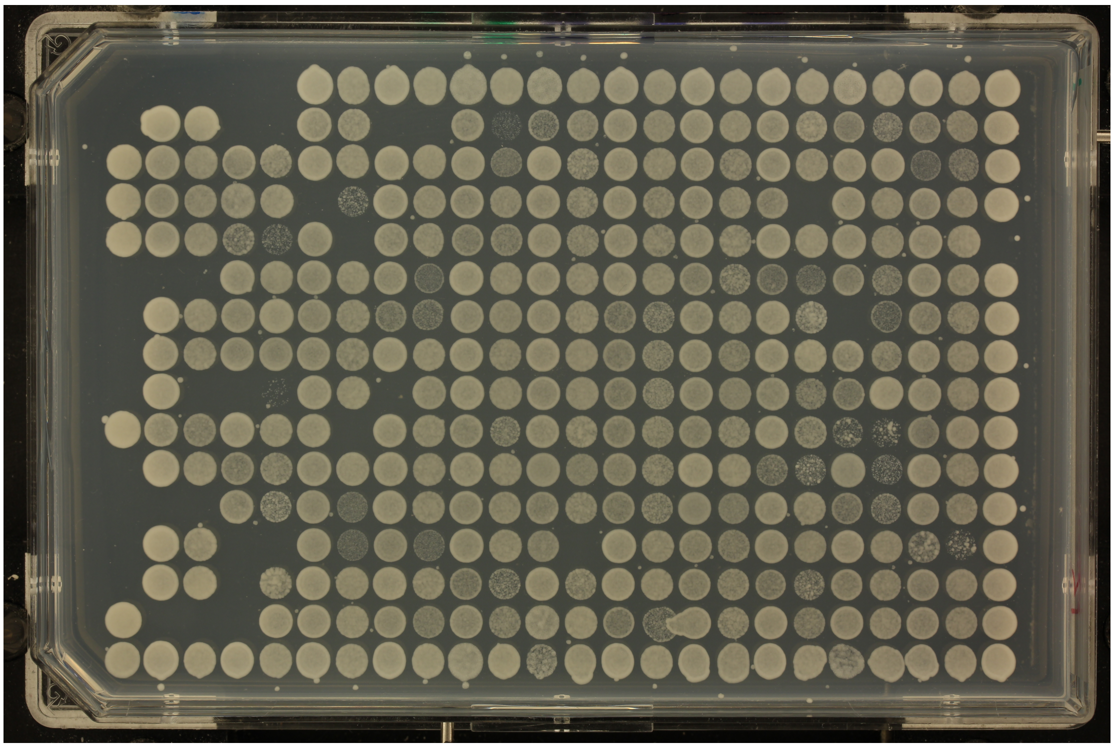


Figure 8.7: 384-spot format plate provided by Colonyzer (Lawless *et al.*, 2010)

the antibiotic resistance in two different drugs will replace the same ORF so at the end the mutated part of DNA will only contain the one cassette and will only have the trait for antibiotic resistance to one of the drugs. Consequently, during the incubation of the mutated strains on the solid agar plates with the two antibiotics, the cells will not have resistance to both antibiotics and will eventually either not survive or present very low growth.

The multiplicative model mentioned in Section 8.2.3 is applied in the miniQFA experiment as well. Suppose that *cdc13-1 yfg1Δ* has a fitness x and *cdc13-1 yfg2Δ* has a fitness y, if yfg1 and yfg2 have independent functions, according to the multiplicative model *cdc13-1 yfg1Δ yfg2Δ* should have an expected fitness of $x \times y$. If yfg1Δ and yfg2Δ interact, the resulting fitness will deviate from the expected $x \times y$, which implies that this pairwise deletion exhibits epistasis.

8.4 Statistical modelling

Both QFA and miniQFA take place multiple times. This repetitive nature of the experiments suggests that hierarchical modelling techniques should be considered while trying to estimate the parameters of the model related to the identification of genetic interactions. Hierarchical modelling will let us account for within ORF variability during the multiple screens and between ORF variability across all ORFs and multiple screens. In this section we will first present a simple approach that was used in the frequentist framework in earlier studies and use that model as a base to build and improve a Bayesian model.

8.4.1 A simple approach to modelling QFA

As suggested by Heydari *et al.* (2016) starting with the simplest version of the model considering random effects and inclusion of all the interaction terms, the model is specified as:

$$f_{clm} = \mu_c + Z_l + \gamma_{cl} + \epsilon_{clm} \tag{8.2}$$

$$\mu_c = \begin{cases} \mu + \alpha, & \text{if c = 0} \\ \mu, & \text{if c = 1} \end{cases} \qquad \gamma_{cl} = \begin{cases} 0, & \text{if c = 0} \\ \gamma_l, & \text{if c = 1} \end{cases}$$

$$Z_l \sim N(0, \sigma_Z^2)$$

$$\epsilon_{clm} \sim N(0, \sigma^2),$$

where,

- c identifies the condition for a given orfΔ. It takes the value 0 for the control strain and 1 for the query,
- m is an index of an orfΔ replicate (denotes the repetition),
- $l = 1, ..., L$ is an index of each orfΔ,
- f is a measure of log-fitness.
  f_{clm}= log(F_{clm}+1); in the log-scale the multiplicative model becomes additive. F_{clm} are the observed fitnesses.
- Z_l represents the fitness of the orfΔ l. It is the random effect that allows us to account for between orfΔ variation by estimating a single σ_Z^2.
- γ_{cl} is an estimate of genetic interaction for the orfΔ l .

One disadvantage of the model above is that we assume constant variance for all the orfΔs. Other obstacles such as high computation and time demand for modelling variances at the orfΔ level and potential restrictions in increasing the complexity of the model in the frequentist paradigm led to adopting the Bayesian logic.

8.4.2 Bayesian hierarchical approach for modelling QFA

As Heydari *et al.* (2016) describe, the above model does not partition variation into population, genotype and orfΔs levels. Furthermore, in the random effects model of the previous section, orfΔ level variation of the multiple repetitions cannot be modelled efficiently since the model assumes constant variance for all orfΔs. In addition, the large number of random effects results in large matrices when the model is fit to data using typical frequentist methods such as maximum likelihood, manipulation of which would have high computational demands and would render calculations time consuming if not impossible. Therefore, they introduced a Bayesian hierarchical model that can account for all sources of experimental variation and incorporate prior beliefs based on conclusions from previous experiments.

When modelling genetic interactions, it is very important to identify which of those are significant and must be included in the model. Therefore in this part we will present a Bayesian model that includes variable selection indicators based on the Kuo and Mallick (1998) method for selecting suitable predictors for the model. This concept has already been introduced in genetics in papers by Avery and Wasserman (1992) and Phenix *et al.* (2011). In the latter paper, in order to identify and quantify causal interactions between genes a strict "ON or OFF" signal is being used, denoting the presence or absence of interactions without allowing intermediate levels of activity. In our case, we incorporate this concept by denoting as δ_l the variable selection indicator for the l-th orfΔ. The indicator is equal to 1 if the corresponding interaction strength is included in the model and to 0 if it does not. So, after incorporating priors that reflect experts opinions about specifications, model 8.2 becomes (Heydari *et al.*, 2016)

$$\begin{aligned} F_{clm} &\sim N(\hat{F}_{cl}, \tau_\epsilon) \\ \hat{F}_{cl} &= exp(f_{clm}) \\ f_{clm} &= \mu + \alpha_c + Z_l + \delta_l \gamma_{cl} + \epsilon_{clm} \\ \mu &\sim N(0, \tau_0^{-1}) \end{aligned} \tag{8.3}$$

$$\alpha_c = \begin{cases} 0, & \text{if c = 0} \\ \sim N(0, \tau_0^{-1}), & \text{if c = 1} \end{cases} \qquad \gamma_{cl} = \begin{cases} 0, & \text{if c = 0} \\ \sim N(0, \tau_\gamma^{-1}), & \text{if c = 1} \end{cases}$$

$$\begin{aligned} Z_l &\sim N(0, \tau_Z^{-1}) \\ \epsilon_{clm} &\sim N(0, \tau_\epsilon^{-1}) \\ \tau_\epsilon, \tau_Z, \tau_\gamma &\sim Ga(\alpha_\tau, \beta_\tau) \\ \delta_l &\sim Bern(p_\delta). \end{aligned}$$

Variation, denoted by the scale parameter of the normal distribution, is now expressed in terms of precision. The equivalent DAG for 8.3 is presented in Figure 8.8.

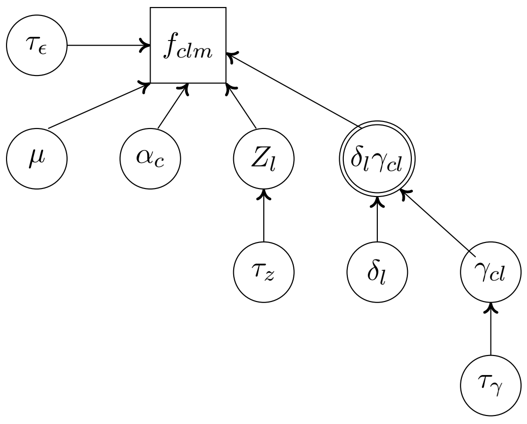


Figure 8.8: DAG for the model 8.3 including variable selection indicators for genetic interaction.

The Bayesian model 8.3 uses variable selection indicators and prior distributions for the parameters of interest. Our prior belief about the parameters that are systematically distributed is expressed through a Normal distribution and Gamma priors are assigned to precisions. In addition, a Bernoulli distribution is used for the variable selection indicator with p_δ probability of success, which reflects our prior belief about the inclusion probability of the interaction parameters in the model.

8.4.3 Bayesian modelling of the miniQFA

For the miniQFA we want to identify the significant genetic interactions between telomeres that show capping defects and simultaneous deletion of two non-essential genes. Based on the previous two models (8.2 and 8.3) and assuming that l represents the ORF deleted from the background screen and l' the ORF deleted from the query screen, we define the following model (Newman, 2017):

$$\begin{gathered} f_{clm} = \mu + Z_l + Z_{l'} + \delta_{ll'}\gamma_{ll'} + \epsilon_{clm} \\ \mu \sim N(0, \tau_0^{-1}), \end{gathered} \tag{8.4}$$

$$Z_l = \begin{cases} 0, & \text{if l = 1} \\ \sim N(0, \tau_Z^{-1}), & \text{otherwise} \end{cases} \qquad \delta_{ll'} = \begin{cases} 1, & \text{if l = l'} \\ \sim Bern(p_\delta), & \text{otherwise} \end{cases}$$

$$\begin{aligned}
Z_l &\sim N(0, \tau_Z^{-1}) \\
\epsilon_{clm} &\sim N(0, \tau_\epsilon^{-1}) \\
\tau_\epsilon, \tau_Z, \tau_\gamma &\sim Ga(\alpha_\tau, \beta_\tau) \\
\gamma_{ll'} &\sim N(0, \tau_\gamma^{-1}),
\end{aligned}$$

- l denotes that ORF l is deleted first from the screen (query deletion) and l' that ORF l' is the second deletion. Z_l and $Z_{l'}$ are the corresponding expected fitnesses. ORF $l = 1$ for the neutral deletion, *HIS3*, and Z_{HIS3}=0 so all the other ORF fitnesses are contrasts from the neutral deletion.
- $\gamma_{ll'}$ represents the genetic interaction strength between the two deleted ORFs.
- $\delta_{ll'}$ is the Kuo and Mallick variable selection indicator that determines whether there is a significant genetic interaction between the two deleted genes and the telomere cap defect.

For biological reasons it is expected that the order of gene deletion should not impact the fitness. So, it is anticipated that the fitness of *cdc13-1yfg1Δxyz2Δ* should be similar to *cdc13-1xyz2Δyfg1Δ*. However, early experiments showed that this is not always valid (Newman, 2017). Therefore, variations of the model consider different levels of symmetry in the effects, as will be further described in Section 9.2.

The miniQFA experiment is of our particular interest since it constitutes the case study on which we apply the Bayesian hierarchical modelling techniques developed and examined in Chapters 3 and 6. Variations of models that could effectively describe the miniQFA process can be explored. In this project we consider a two-way Anova structure with random main and interaction effects. The main effects represent the two deleted ORFs and their interactions represent the equivalent genetic interactions between the genes of interest. In addition, the examined models take into account potential symmetry in the effects. We approach variable selection using the Kuo and Mallick method since in Chapter 6 we concluded that it appears to be more efficient for the type, structure and size of the models that we study.

In the following chapter we will give an overview of the resulting dataset of the miniQFA experiment, as well as present the specifications and results of various models explored and built aiming to identify genetic interactions between the genes of interest related to telomere capping defects.

Chapter 9

Case study: miniQFA – Yeast robotic genetics

Highlighting important gene interactions in *Saccharomyces cerevisiae* can lead to understanding of how groups of genes connect and contribute to telomere length maintenance in this model organism, with the potential to extend this knowledge to more complex eukaryotes. The miniQFA experiment presented in Section 8.3 aims to identify epistatic relationships between telomere defects and pairs of gene deletions. So, in this chapter, firstly we will provide an overview of the dataset deriving from this experiment. Then, we will focus on the specifications of the Bayesian hierarchical modelling methods used to identify genetic interactions between genes that carry the mutation *cdc13-1* and combinations of simultaneous deletion of two genes, coming from a gene deletion library, that could potentially affect telomere capping. Finally, we will present the results from various models that concentrate on identifying significant interactions and show whether an existing telomere capping defect is being suppressed or enhanced by the absence of particular combinations of genes.

9.1 miniQFA dataset

The dataset, resulting from the miniQFA workflow, includes information about the experiment such as identification of the yeast colony by its exact position on the agar plate, the experiment number and date. It also contains information about the conditions under which the strains grow, such as the agar growth medium, that is designed to support the growth of the micro-organisms and cells inoculated, and the temperature at which the experiment took place. In this experiment temperature can take two values, either 27^oC or 33^oC, due to the nature of temperature sensitive mutations applied to genes related to telomere capping as discussed in Section 8.2.2.

Measurements of fitness, that are actually cell density estimates, is another important attribute provided in the dataset. Measures include the growth rate (r), the carrying capacity (K) which is the number of a species that an environment can sustain, as well as transformations in the log-scale and products. Previous research has shown that when modelling QFA some interactions were only identified when K was used for measuring fitness and not when other measurements were considered (Heydari, 2014). Therefore, it is important to explore alternative models, using various measurements of fitness as the response, and compare them by using statistical criteria such as the deviance information criterion (DIC), as well as using knowledge about some already discovered interactions that belong to known complexes. For each observation there is also information about the pair of deleted genes. Equally important for the development of the model is the knowledge about the background screen which is provided in the dataset. In some cases the mutation *cdc13-1* is applied, whereas other experiments use wild-type *CDC13+* yeast strains. A sample of the raw data is presented in Figure B.1 of the Appendix.

9.2 Statistical modelling specifications

The aim of this case study is to identify, through Bayesian hierarchical modelling, existing genetic interactions between genes of interest that are deleted and the particular mutation *cdc13-1* that is known to affect telomere length. The resulting model contains estimates for the strength of interaction effects, as well as the main effects of each gene separately and of other important unknown parameters, such as the mean and various precisions that constitute measurements of variation.

The models built for the yeast genome data are based on the statistical background established in Section 3.3.2, where we explored variable selection on a non-standard hierarchical two-way Anova model with various cases of asymmetry in the main and interaction effects. According to the results of the simulation study presented in Chapter 7, where we examined both the Kuo and Mallick approach, and the Horseshoe priors, we concluded that the first method appears to be more efficient for applying variable selection to the interactions of the particular models of interest. Therefore, we follow this approach to model the miniQFA, which is is treated as an extension of a two-way Anova model with random effects, in accordance with the models explored in the simulation study (Section 7.3).

As we saw in Section 3.3.2.1, the model for the case where both the main and the interaction effects are asymmetric is defined as:

$$X_{ijk}|\mu, \alpha_j, \beta_k, I_{jk}, \theta_{jk}, \tau \sim N(\mu + \alpha_j + \beta_k + I_{jk}\theta_{jk}, \tau^{-1})$$

where $i = 1, ..., N_{jk}$, with N_{jk} being the number of observations for the group with α_j

and β_k. In addition, $j = 1, ..., n_\alpha$ and $k = 1, ..., n_\beta$, with n_α representing the number of levels for the first and n_β the number of levels for the second categorical variable. For the miniQFA, the numbers of levels represent the numbers of different genes in each of the two gene groups. The response variable is the fitness measure, that is already available in the dataset, and measures the growth of the yeast colonies. The main effects are grouped in two categorical variables. The first variable (α_j) is related to the first gene deletion and the second (β_k) refers to the second group of orf*Δ*s. The interactions in the model represent the genetic interactions between the examined genes from the first and the second groups. Variable selection is applied on these interactions, in accordance with the logic followed throughout the previous relevant chapters, in order to distinguish between the ones that appear to be more meaningful for the model and the ones that are less significant.

Various levels of asymmetry in the effects are also considered in the model. The levels of asymmetry in the main effects and interactions accounted for throughout the development of the modelling logic have a biological origin. For biological reasons, it is expected that ORF fitness distributions behave similarly regardless of whether a gene of interest is deleted first or second. So, it is expected that the fitness of *cdc13-1yfg1Δxyz2Δ* should be similar to *cdc13-1xyz2Δyfg1Δ*. Model variations using symmetric main effects and interactions reflect this idea. For symmetric main effects, the fitness of the two deleted ORFs come from the exact same distribution and gene library. In addition, for symmetric interaction effects the variable selection indicator and the effect of the interaction *yfg1Δxyz2Δ* are equal to those of *xyz2Δyfg1Δ*. However, early experiments showed that this assumption does not always hold (Newman, 2017). To account for this discrepancy, models with asymmetric effects for both main and interaction terms are explored. In this case, the main effects come from a different distribution and gene library and the effect for the interaction *yfg1Δxyz2Δ* is different from the effect for *xyz2Δyfg1Δ*. A final combination for symmetry that is reasonable for this experiment is the asymmetric main effects with symmetric interactions. In this variation we assume that the sequence in which the genes of interest will be deleted plays an important role in the resulting fitness, nevertheless their interactions are the same regardless of which gene is deleted first.

The resulting interactions are graphically illustrated through genetic interaction plots (GIS plots), an example of which is presented in Figure 9.1. These plots contain thousands of interactions, often difficult to interpret, therefore they are searchable, and supplementary tables and lists of pairwise deletions are usually used to facilitate demonstration of results. GIS plots display how the observed fitness deviates from the predicted fitness under the assumption of no epistasis. The labelled interactions show the most significant deviations and correspond to interactions with inclusion probability greater than 0.7. This probability was selected arbitrarily as a pragmatic choice. The horizontal axis represents the computed ‘predicted fitness’ assuming no interactions occur between the examined genes. The values are calculated by adding the individual main effects and the overall mean. These values reflect Fisher's multiplicative model of epistasis assumption for what impact on fitness is expected when each gene is deleted. The vertical axis represents the fitted value of the fitness measure for each pairwise deletion including the interactions. They are the result of the summation of the individual main effects, their interaction effects and the mean. Points marked with green represent mutations (deletions of two genes) that cause phenotypic enhancement. Thus, even less yeast strains that already have a telomere capping defect, such as *cdc13-1*, survive in the absence of the two genes of interest. Equivalently, mutations marked with red suggest suppression of the telomere capping defect. This implies that with the absence of these genes the mutant cultures grow more than they would without this pairwise deletion. For both the phenotypic enhancement and suppression categorisation we consider interactions with inclusion probability greater than 0.7. We denote with blue the deletions that show no interaction with the telomere capping defect. We assume that interactions with inclusion probability less than 0.7 fall in this category. These points lie close to the line of equal growth, denoted with a dashed grey line. Double mutants of the form *yfgΔyfgΔ*, with a repeated deletion are marked in purple. They are expected to be unviable, and hence have zero or very low fitness. Deviation from double-l mutants' expected low value might indicate that neighbouring strains have occupied the space of the colonies that did not survive (Newman, 2017). This effect is investigated by Boocock and Lawless (2017) that describe a mechanistic model of the diffusion of nutrients through agar.

The knockout library for the first gene deletion consists of 113 genes and for the second of 143 different genes. For the models that incorporate symmetry in the main effects, we assume that both gene deletions derive from the biggest library. It is worth mentioning at this point that, for the statistical modelling, numerical values are assigned to gene names deriving from two gene libraries. For the models with symmetric main effects, only the biggest gene library is used, resulting to "missing" levels for the first categorical variable. This has already been considered in the simulation study (Section 7.1), in which we ensured that the code can handle "missing" levels.

Previous research has shown that the genes POT1 and DPH5 cause unexpected behaviours, therefore it is suggested that they are removed (Newman, 2017). Consequently, these genes were discarded from the library and the corresponding observations were removed from the dataset. Various models at different experimental temperatures, either 27^oC or 33^oC, with varying fitness measurements can be explored. For each model a subset of the original data is selected, according to specific experimental conditions examined. In all cases the resulting subset of data involves approximately 83,000 observations.

9.3 Results from statistical modelling

Different combinations of model specifications and experimental conditions lead to variations of models related to inferring genetic interactions. In Section 6.2.4 it was shown that the parallel version of TWIiS, for the particular type of modelling that we examine, performs better for problems with increased complexity in terms of number of levels of the categorical variables and sample size. Therefore, for speed and efficiency this version was used to build the models explored and presented in this section. Their results and effects on telomere capping are illustrated through GIS plots. All results derive from one million iterations of the Gibbs sampler with a thinning factor of one hundred. The scientific conclusions drawn in this section are the result of a single run, however MCMC runs were repeated multiple times from different random starting points to ensure that the results are consistent. In consistence with the logic followed in the simulation study (Chapter 7), diagnostic tests were conducted to assess the posterior behaviour of the yeast genome modelling parameters. More precisely, we look at the traceplots of the estimated parameters to assess mixing, the autocorrelation plots that should follow a decreasing tendency and the univariate marginal posterior plots. Samples of these diagnostic plots for the following models are presented in Appendix B.2 and relevant references are available in the sections of the models that follow.

In the following sections we will explore various models that fit to subsets of the miniQFA dataset with particular specifications of interest. We will focus on using the growth rate r as a measure of fitness since it was considered an important fitness measurement in previous relevant research.

9.3.1 Case study 1: Using the growth rate (r) as measure of fitness and *cdc13-1* mutant strains

The first case study considers using the growth rate r, as a measure of fitness for *cdc13-1* mutant yeast strains. The two-way Anova model with random effects fitted on the subset of the original dataset with the desirable characteristics includes variable selection using the Kuo and Mallick approach on the interaction effects. We explore the fitness behaviour at two different temperatures 33°C and 27°C to see how incubation conditions affect genetic interactions.

In miniQFA the background screen consists of the yeast strain BY4741, with additional mutations applied for the synthetic genetic array (SGA) technique described in Section 8.2.1 and the temperature sensitive mutation *cdc13-1*. The query screen is considered to be the first gene deletion and a second gene deletion is additionally applied, resulting in the pairwise deletion examined. Therefore, the model can also be viewed as a three-way Anova, where the mutation can be considered as the first categorical variable with one level.

9.3.1.1 Temperature: 33°C

The first model explored in this case assumes that both the main and interaction effects are asymmetric, and involves *cdc13-1*. The corresponding GIS plot is presented in Figure 9.1. To see the impact of accounting for symmetry in the effects, a second model was fitted that treated both the main and the interaction effects as symmetric. The resulting GIS plot is displayed in Figure 9.2. Finally, we explored an additional model to see how epistatic genetic interactions are formed in the case where only the interaction effects are considered to be symmetric. The GIS plot illustrating these relations is available in Figure 9.3. The posterior results and diagnostics were checked for all the cases. Indicatively some traceplots and autocorrelation plots are available in Appendix B.2.

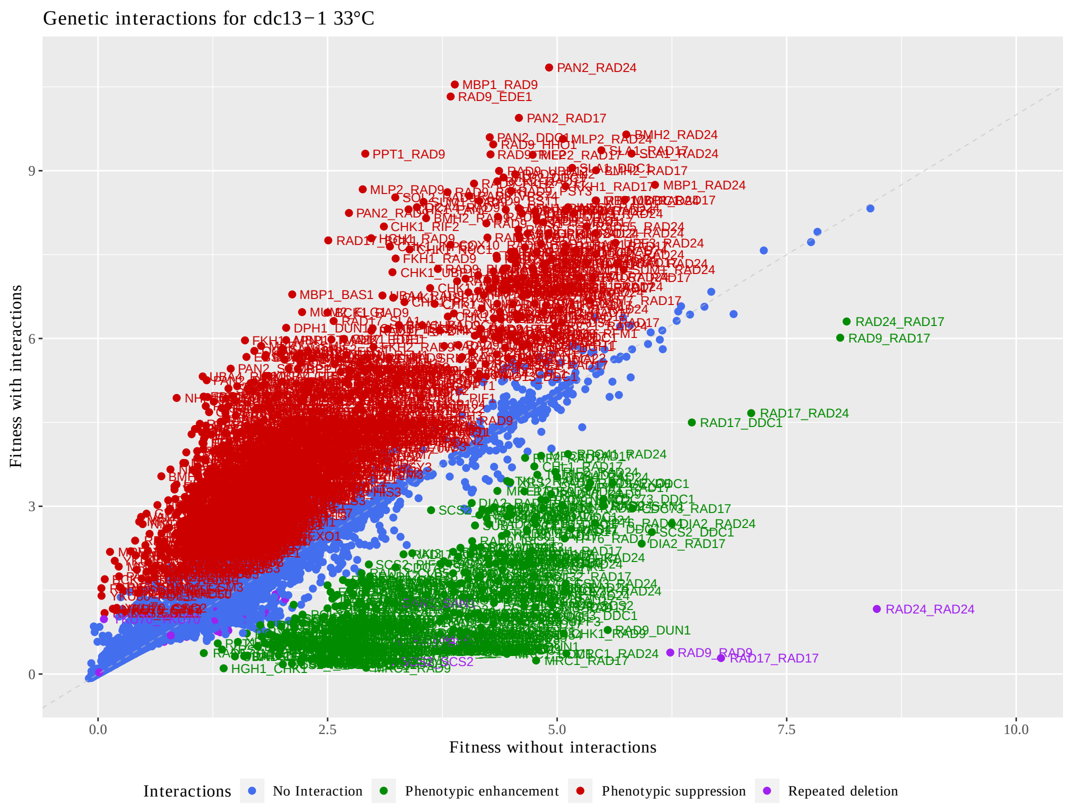


Figure 9.1: Genetic interactions plot for experiment: *cdc13-1* at 33°C with r as measure of fitness. Both main and interaction effects are asymmetric.

These plots as expected show that there is vaguely a linear relationship between the variables and therefore a linear model was correctly assumed to be appropriate for fitting the data. In addition, GIS plots are anticipated to have three main areas of concentrated

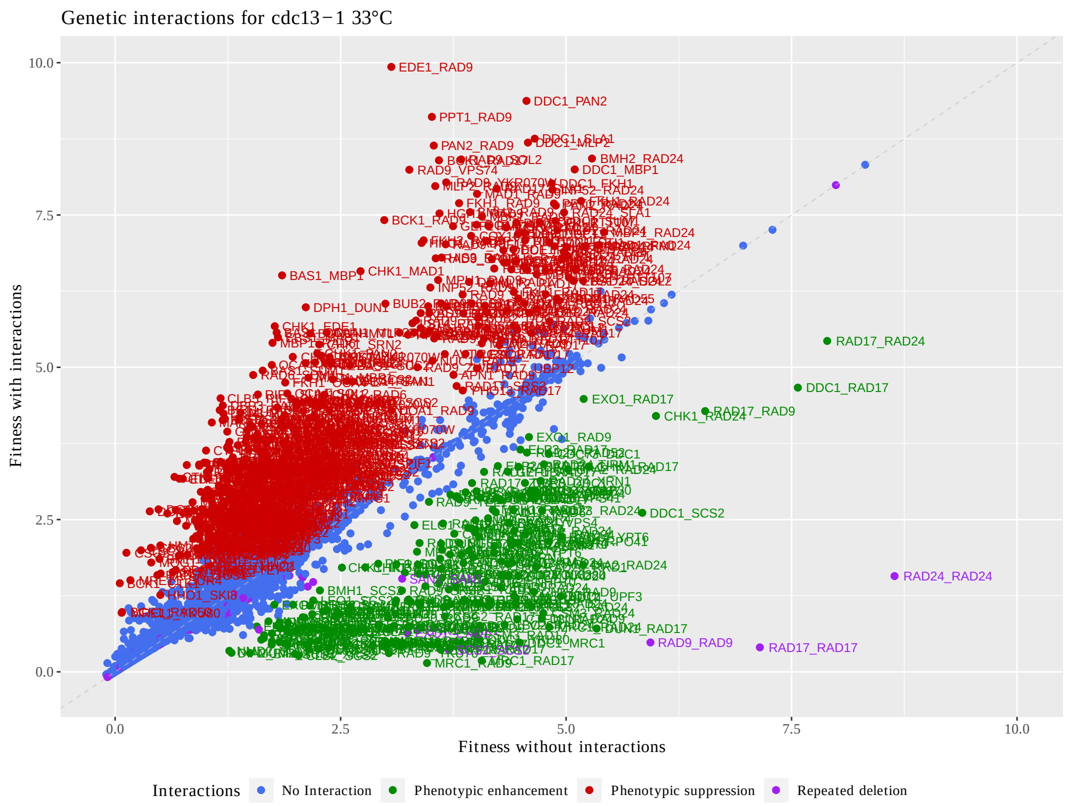


Figure 9.2: Genetic interactions plot for experiment: *cdc13-1* at 33°C with r as measure of fitness. Both main and interaction effects are symmetric.

points. One area is above the line of equal growth for the majority of the interactions that cause phenotypic suppression, one mass below the line for the enhancement and one on the line of equal growth for the interactions that appear to not affect the telomere capping defect. Furthermore, additional interactions from each category are expected to be spread further away from the majority indicating stronger impact on the background screen. This is valid for the models that treat the effects as either both asymmetric or both symmetric as shown in plots 9.1 and 9.2 respectively. The last model accounting for the assumption that only the interaction effects are symmetric appears to slightly deviate from this notion and additionally presents negative fitted values for some interactions. Double repeated deletions indicated in purple are expected to show low observed fitness as shown in Figures 9.1 and 9.2 as usually these mutant strains are not able to survive. Higher values can also be observed if neighbouring growing colonies on the agar plate occupy the spot of repeated deletion mutants, and therefore they are falsely categorised.

Regarding the last model, one disadvantage of presenting its results using a GIS plot is the fact that since the labels are named after the interactions and the interactions

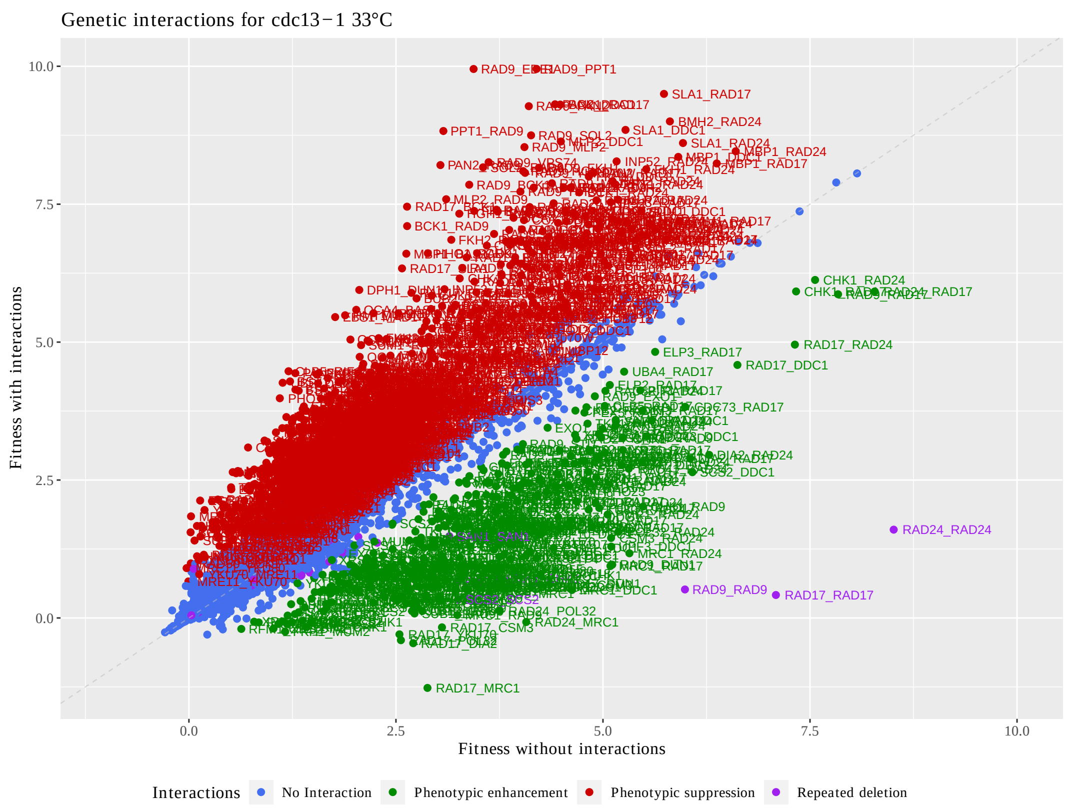


Figure 9.3: Genetic interactions plot for experiment: *cdc13-1* at 33°C with r as measure of fitness. Only the interaction effects are symmetric.

are symmetric there should be only one label per interaction as the order is not important. However, the points on the graph also include information about the main effects, therefore, in order to be able to distinguish between the sequence of gene deletions two different labels are used. For example, for the point RAD17_RAD24 the y-axis shows the value of adding μ to the main effect estimate for RAD17 which was the first deletion, the main effect for RAD24 which was the second deletion and the interaction effect for RAD17_RAD24. Similarly, the point RAD24_RAD17 has a fitted value calculated from adding μ to the main effect estimate for RAD24, the main effect for RAD17 and the interaction effect for RAD17_RAD24, since for symmetric interactions we assume that only one interaction exists for this combination of genes and implementationally we set the estimate for RAD24_RAD17 to be equal to RAD17_RAD24. Then, the label RAD17_RAD24 would be presented twice in the plot without further information about the sequence of deletion that matters for the main effects.

In this case study three different models were fitted on the same dataset. The decision regarding the model that best describes the biological phenomenon examined is multifactorial. Apart from the expected behaviour reflected on the GIS plots mentioned earlier and the symmetry assumptions presented in Section 9.2, from a biological perspective, simpler and sparser models are preferred. All models estimate a similar posterior inclusion probability for the interactions, with the model treating both the main and interaction effects as symmetric having $p = 0.19$, slightly better than its alternatives (both $p = 0.22$). We will look at the estimated interactions per category for the models with both main and interaction effects as symmetric and both asymmetric. Table 9.1 summarises the interaction classification from both models and the number of common elements per category. As expected the model with both the main and interaction effects as symmetric identifies fewer enhancements and suppressions leading to a model with a decreased number of interactions. Please note that a posterior inclusion probability $p = 0.22$ indicates that a subset with approximately 20% of the interaction variables is considered to be important and should be included in the model. However, for the genetic experiment and presentation of results we further restrict the interaction significance and we categorise as causing phenotypic enhancement or suppression the interactions that have inclusion probability greater than 0.7. Hence, the ratio of the important interactions over their total number, as presented in the table, is less than the posterior mass.

Table 9.1: Classification of interactions per category from models with both main and interaction effects being asymmetric versus symmetric.

Category	Both asymmetric	Both symmetric	Common
Repeated deletion	111	111	111
No interaction	14,230	9,127	6,508
Phenotypic enhancement	322	208	91
Phenotypic suppression	988	535	252

Tables that present the top 20 suppressor and enhancer combinations for these three scenarios, ranked by the strength of the estimated genetic interaction are available in Appendix B.2 (Tables B.1 and B.2).

From a statistical point of view, we presented in Section 2.5 some alternative ways used in the Bayesian framework for model comparison. One criterion which is easy to estimate when using MCMC and often encountered in the literature is the deviance information criterion (DIC). It uses the log-likelihood to calculate the deviance, a goodness-of-fit statistic, and penalises models with large numbers of parameters in favour of sparser models. Models with smaller DIC are preferable. Table 9.2 displays the DIC values for the three models examined in this section. All values are close, precluding drawing firm conclusions. Besides, information from predictive criteria is usually combined with field-specific knowledge. Therefore, conclusions about the preferred model should be made after further biological exploration of the results.

Table 9.2: Deviance information criterion for models regarding mutant *cdc13-1* yeast strains with growth rate (r) as fitness measure at 33°C

Case	DIC
Both asymmetric	299,682.8
Both symmetric	306,529.4
Symmetric interactions	301,414.3

In genetics, biological pathways are series of actions that trigger molecular events leading to creation of molecular products or changes in the state and processes of a cell. Proteins can contribute to more than one biochemical pathway along with other proteins, forming complexes. Regarding *S. cerevisiae* there are some known protein complexes contributing to mechanisms such as DNA damage repair and telomere maintenance. MiniQFA aims to extend this knowledge through identification of additional pairwise interactions. The Saccharomyces Genome Database (2020) provides detailed lists of explored interactions resulting from various studies on this microorganism. Regarding the evaluation of the actual interactions classification, study of the GIS plots for both the model with asymmetric main and interaction effects and the symmetric model, and cross-validation of their results with existing interactions in the database, proved that many known relations between genes are indeed among the model estimates. However, not all known interactions are expected to be present in the plot, since they are not all involved in pathways related to telomeres.

One example of epistatic interactions based on the results of the modelling process involves the gene RAD24. According to the European Bioinformatics Institute (EMBL-EBI, 2020), Rad24 is a checkpoint protein that forms a complex with replication factor C (RFC) proteins, further loading the Rad17-Mec3-Ddc1 complex, which is also called “sliding clamp”. This complex detects DNA damage and sends signals to halt progression of the cell cycle and activate the repair mechanism. Looking in Figures 9.1 and 9.2 we can see that in the region illustrating the interactions that enhance the telomere capping defect, there are two pairwise deletions “RAD24_RAD17” and “DDC1_RAD17” that reflect this protein cooperation, showing that the model identified correctly proteins involved in the same pathway. Their presence and position in the plot further implies that their pairwise absence affects, and in fact enhances, the *cdc13-1* telomere capping defect.

On the other hand the pairwise deletion “EDE1_RAD9” is situated above the dashed line of equal growth and appears to suppress the telomere capping defect. Even though there is not available research in the literature at the moment to assess this potential interaction, Ede1 is a protein involved in “cytokinesis”, which is the process of cell division. Rad9 participates in protein complexes that are related to DNA damage, such as the Rad9-Hus1-Rad1 (9-1-1) complex which is a sensor for DNA damage, therefore, a possible relation is plausible, and so potentially could be subjected to further biological research.

9.3.1.2 Temperature: 27°C

Temperature sensitivity plays an important role in the outcome of the miniQFA experiment as yeast colonies with certain mutations and antibiotic resistance behave differently depending on the incubation conditions. One way to point out this difference is through illustrating, in Figure 9.4, the GIS plot for the model that accounts for both the main and interaction effects as asymmetric, fitted on the dataset where the measure of fitness is still the growth rate and the background screen is *cdc13-1* mutant yeast strains, but the experimental temperature is 27°C. Comparing this GIS plot to the equivalent Figure 9.1 for 33°C, we can see that some of the most distinguishable interactions involve different genes. For example the interaction "PPH3_RAD24" is identified as causing phenotypic suppression of the telomere capping defect in both cases but is further away from the mass in the experiment at 27°C. According to the Saccharomyces Genome Database (2020), PPH3 regulates recovery from the DNA damage checkpoint.

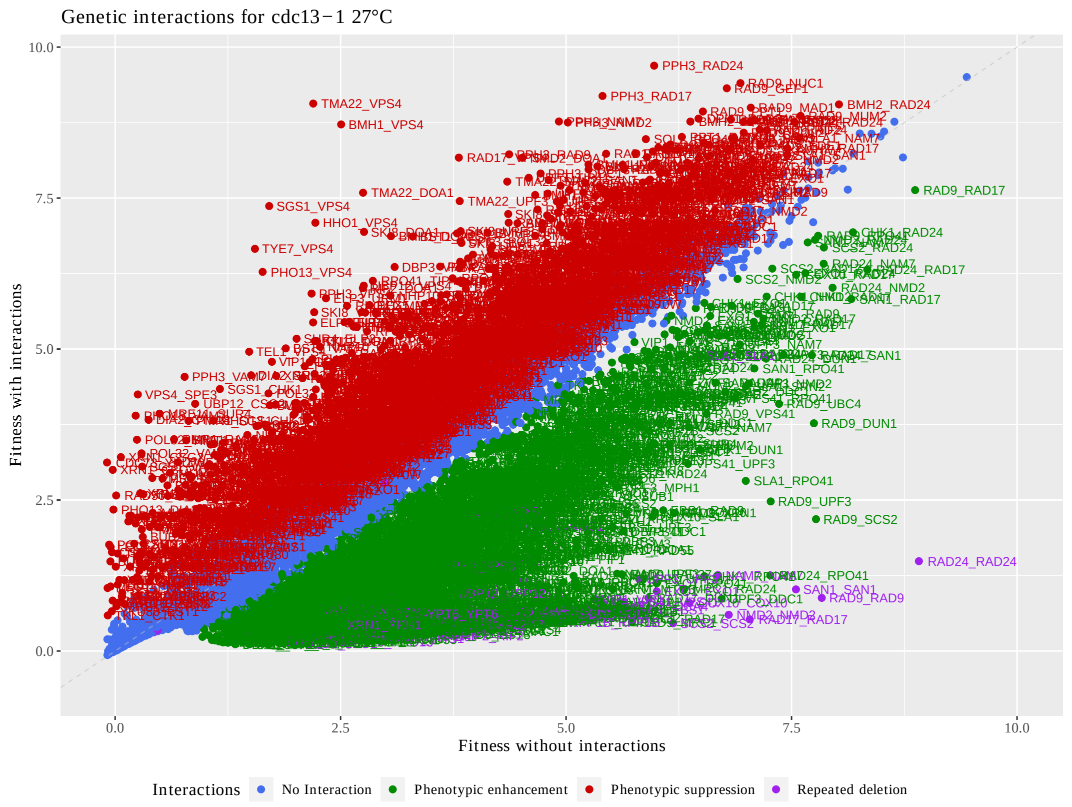


Figure 9.4: Genetic interactions plot for experiment: *cdc13-1* at 27°C with r as measure of fitness. Both the main and interaction effects are asymmetric.

It would be interesting to see how many interactions per category are mutually identified from these two models at different temperatures. Table 9.3 summarises these results and points out that most of the interactions estimated at 27°C that appear to have no impact on the telomere defect are also identified by the model fitted on the dataset for the experiment at 33°C. The categories with phenotypic impact show greater divergence, with the model at 27°C identifying more pairwise combinations to have a considerable impact on the scientific question that we explore.

Table 9.3: Classification of interactions per category from the models with both main and interaction effects being asymmetric at 27°C and 33°C.

Category	Both asymmetric at 27°C	Both asymmetric at 33°C	Common
Repeated deletion	111	111	111
No interaction	12,086	14,230	11,276
Phenotypic enhancement	1,891	322	167
Phenotypic suppression	1,563	988	308

As in the previous cases, to assess the posterior output we looked at the traceplots of the estimated parameters, which showed good mixing, the autocorrelation plots that had a decreasing tendency and the univariate marginal posterior plots. Indicatively we present some of these diagnostic plots in Appendix B.2 (Figure B.3).

9.3.2 Case study 2: Using the growth rate (r) as measure of fitness and *CDC13+* wildtype strains

Another model that has biological interest is related to the subset of observations for which the wildtype *CDC13+* yeast strain was used as a background screen. Two genes of interest are also deleted as in the case of the *cdc13-1* mutants and the growth rate is used to measure fitness. Now, the interactions are not in the background of a telomere capping defect, but they are still relevant to telomeres and potentially participate in biological pathways and protein complexes related to them, since most of the genes selected for the miniQFA showed some connection with these chromosomal structures. The classification in this case shows whether the expected fitness for the pairwise deletion examined is greater or less than the actual fitness, meaning that the absence of these genes had a negative or positive impact in yeast culture growth respectively. Similarly to the previous models we looked at the posterior diagnostic plots of the estimated parameters, examples of which are available in Appendix B.2 (Figure B.4).

The GIS plot for the model that treats both the main and the interaction effects as symmetric is available in Figure 9.5. Among the genes that appear often are COX10, CTK1 and CBC2. According to the Saccharomyces Genome Database (2020) the first is involved in the biosynthesis of *heme A*, a molecule vital for oxygen carrying. CTK1 and CBC2 are related to RNA regulation and RNA cap-binding respectively. Their potential impact along with their paired genes to telomeres could be further biologically explored.

Comparison of the mutually identified interactions and classification of this model and the equivalent for the *cdc13-1* mutant yeast strains is available in Table 9.4. Most of the pairwise deletions that appear to present no interaction in the model that we have examine are common for both models. However, there are not many common interactions for the deletions that appear to have a negative or positive impact on growth and the equivalent model for the mutants with respect to the values for phenotypic enhancement and suppression, that can alternatively be seen as showing whether the deletion had a negative or positive effect in the fitness of the yeast cultures studied.

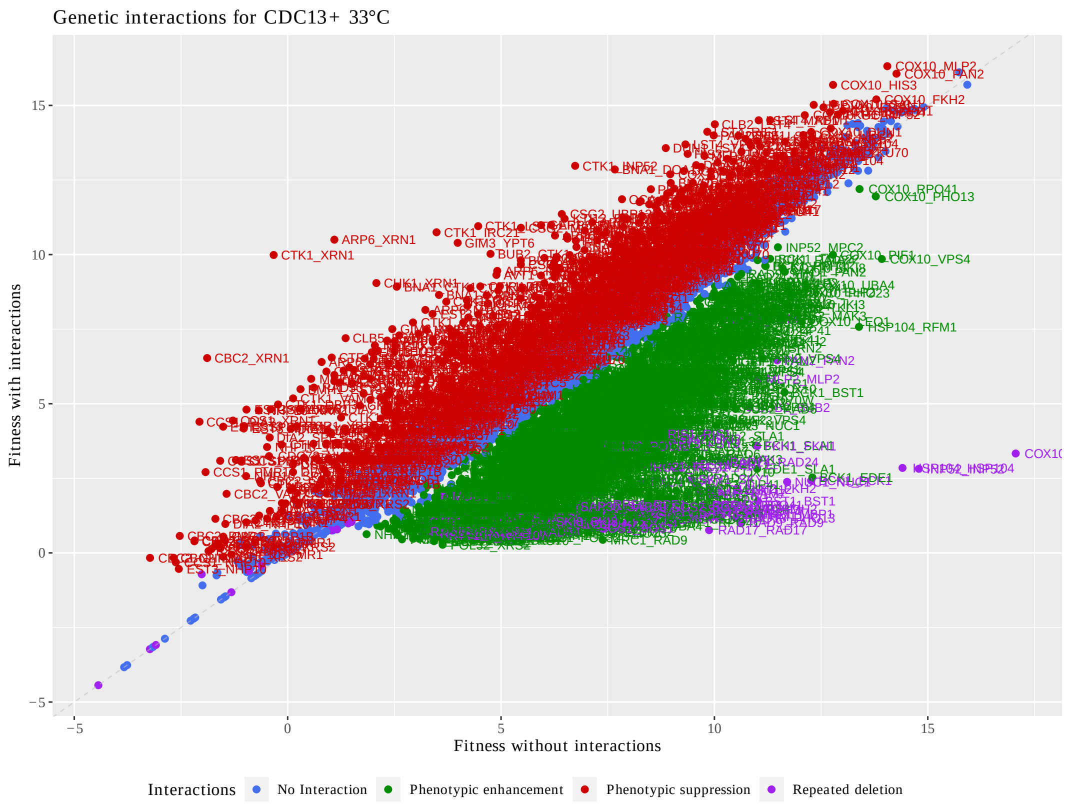


Figure 9.5: Genetic interactions plot for experiment: *CDC13+* at 33°C with r as measure of fitness. Both main and interaction effects are symmetric.

Table 9.4: Classification of interactions per category from the model with both main and interaction effects being symmetric at 33°C using *CDC13+* as background screen.

Category	Both symmetric *CDC13+*	Common with *cdc13-1*
Repeated deletion	111	111
No interaction	7,765	7,193
Negative growth impact	1,280	73
Positive growth impact	825	55

9.3.3 Additional case studies

There exist many combinations of experimental and biological specifications that can be explored through variations of the two-way Anova models with random effects and variable selection that we examine. Apart from the models presented in the previous sections we also fitted additional models that considered alternative fitness measures such as "$r \log K$" and "$\log K$". However, the results suggested a potential non-linear relationship and inappropriateness of the two-way Anova model that we explored, or estimated inclusion probabilities close to 0.4 affecting negatively the sparsity assumption. An example of the resulting GIS plots from using $\log K$ as fitness measure at 33°C for the *cdc13-1* mutant background screen is available in Appendix B.2 (Figure B.5).

Depending on the genetic problem and scientific question, further model variations can be explored and evaluated efficiently using the parallel version of TWIiS. The results presented are for one million iterations of the Gibbs sampler. As we discussed throughout the sections of this chapter where we presented the results from various models, for this number of iterations, the posterior output of the yeast genome modelling parameters showed good diagnostic plots, with the traceplots indicating mixing and the autocorrelation plots following a decreasing tendency.

The results for one million iterations of the Gibbs sampler for the particular models that we use for the yeast genome dataset, with the specific model assumptions that we discussed in earlier sections, were available within approximately three hours (using 8 cores and 32 GB memory). For the same models, dataset and computing resources, benchmarking showed that JAGS needed approximately twenty-nine hours, rendering TWIiS almost ten times faster. The fast processing time and flexibility in adjusting certain model parameters allows scalability of the problem in terms of variables and sample size, and renders quick model exploration feasible.

Chapter 10

Conclusions and further work

In this chapter we present the main objectives, contributions and conclusions of this research project, and discuss some possible directions for future work.

10.1 Objectives and contributions of the thesis

The primary objective of this thesis was to approach the computational and implementational aspect of Bayesian hierarchical modelling with a functional programming perspective and assess whether it improves scalability and performance of the MCMC algorithm used, compared to programs written in more classic probabilistic programming languages. In particular, the model of interest was a two-way Anova with interactions and random effects. Variations of the model accounted for different levels of asymmetry in the effects, with the main and/or interaction effects being symmetric/asymmetric, based on the rationale of the genetic experiment that was used as a case study. Variable selection was applied on the interactions using two main methodologies; indicators and adaptive shrinkage priors. Identification of the technique that appears to be more appropriate for this type of model was an additional aim. The models were fitted using a Gibbs sampler developed in Scala, a programming language that integrates functional and object-oriented features. Assessment of the results and performance was conducted through comparison against JAGS. Additional implementations in Stan and Rainier, both using Hamiltonian Monte Carlo (HMC), were also explored to evaluate the performance of HMC for these models. The final objective of this research project was to apply the developed method in a case study from the field of yeast genetics.

Through the simulation study presented in Chapter 7, we investigated the results and performance of the Gibbs sampler code written in Scala, referred to as TWIiS (Two-way Interactions in Scala), against JAGS, which was developed in R through the library ‘rjags’. Our main contribution was to show that TWIiS is more efficient than the equivalent JAGS code for various levels of model complexity and sample size that were tested. The extent to which TWIiS is more performant than JAGS depends on the model specifications. The metric that was used for efficiency comparison is the effective sample size per second (ESS/sec) that accounts not only for the computational time but also for how well the algorithm mixes. Further efficiency gain for TWIiS was achieved for models with increased number of covariates and observations with the parallel version of the code. Parallelism takes advantage of the multicore technology by spreading some of the computational workload across the available nodes.

In the same Chapter we compared two variable selection methods, the Kuo and Mallick approach and the Horseshoe prior, that were embedded in the Gibbs samplers developed in TWIiS and JAGS. It was also shown that the first methodology appears to be more efficient and appropriate for the special cases of two-way Anova models with random effects that we examine. Variable selection indicators follow discrete distributions, therefore this technique cannot be used along with HMC that only supports continuous variables. Having studied the Gibbs sampler's performance along with the Horseshoe prior, we further explored how this continuous prior method performs with HMC using Stan and Rainier. The results showed that our implementation of a Gibbs sampler appears to be more performant for the type of models that we study, compared to HMC.

The last contribution of this thesis is related to the scientific understanding of the yeast genetic experiment from which the dataset of our case study derives. The miniQFA experiment that we studied aims to identify genetic interactions between genes of interest and specific mutations that cause telomeric defects applied to yeast strains. Using TWIiS to fit Bayesian hierarchical models to subsets of the original dataset with specific experimental characteristics enables fast exploration and identification of interactions that are potentially significant after being subjected to further biological evaluation. This analysis is presented in Chapter 9, which is devoted to the statistical modelling of the yeast genome case study.

10.2 Conclusions

After introducing some key notions of Bayesian inference in Chapter 2, we presented three widely used MCMC algorithms, Metropolis–Hastings, the Gibbs sampler and Hamiltonian Monte Carlo. We also explored variable selection methods in the Bayesian framework and we focused on two categories, variable selection indicators and in particular the Kuo and Mallick approach, and the adaptive shrinkage priors and specifically the Horseshoe prior.

In Chapter 3, we introduced hierarchical two-way Anova models that include two categorical variables, and their interactions on which we applied variable selection. The effects were treated as random, and different levels of symmetry in the effects resulted in four model variations for each variable selection method. The Gibbs sampler, selected to be used for our modelling approach, requires calculation of the full conditional distributions of the unknown parameters. The derivation of the full conditionals for our models of interest was presented in this Chapter. Unlike R where there are libraries, such as 'rjags', that implement a Gibbs sampler, in Scala we had to develop the algorithm from the ground up. Therefore, the full conditionals presented here were used to develop TWIiS, the Scala program for the desired model specification.

Chapter 4 contained a literature review of the existing strategies for scaling Bayesian inference methods that can be computationally expensive. As shown, they can be categorised in two main directions, subsampling and divide-and-conquer strategies. The first category refers to methodologies that use subsets of the initial dataset to estimate expensive calculations such as of the log-likelihood at the accept/reject step of MCMC algorithms. The second involves strategies that take advantage of computing resources and parallelisation to speed-up the sampling process. Most of these methodologies have limited application to hierarchical models, since in this case there are both common and group-specific parameters, therefore we mentioned other approaches that have been developed for this purpose.

Since we implemented the examined methods using functional programming, we presented in Chapter 5 some principles of this programming paradigm that were followed during the development of the code. We also showed how terms and ideas that derive from category theory are embedded in functional programming features to enable code re-usability. Furthermore, we showed that probabilistic programming can be incorporated in the functional framework, and we presented Rainier, that was later explored as an alternative approach to TWIiS and JAGS.

In Chapter 6 we presented some implementational details for TWIiS, JAGS, Stan and Rainier, the alternative approaches considered for the two-way Anova model of interest, with particular emphasis on the first. Particularly for TWIiS we concluded that the choice of the main structure used to store the data has a significant impact on the efficiency of the code. The structure affects the time for search and basic calculations that are repeated multiple times due to the iterative nature of the sampling process. In addition, we explained the logic behind the parallel version of TWIiS and the main differences between the serial and the parallel implementation. Due to the nature of the Bayesian algorithms, only conditional independent variables can be updated in parallel. Therefore, parallelism was mostly applied to update the interaction coefficients. When writing code in Scala, there is flexibility in the degree of functional features that can be incorporated in the code. Benchmarking tests showed that for the serial version, less functional parts of the code, such as mutable structures, were more performant than more functional alternatives. However, converting these parts to functional was necessary in order to avoid concurrency problems during parallelisation. Finally, after exploration of the effect of parallelism on efficiency using benchmarking, we concluded that for smaller numbers of interactions and sample sizes, the parallel version had negative impact on the runtime due to the overhead caused by splitting and combining the workload to and from the worker nodes. As the problem size, in terms of model complexity and data size, increased, the parallel implementation became more efficient than the serial version.

In Chapter 7 we presented the results of the simulation study with two main aims: firstly, to evaluate and assess the results and performance of TWIiS against JAGS, and secondly to determine which of the two variable selection techniques that were explored appear to be more appropriate for the models studied. For both variable selection using indicators and Horseshoe priors, we used three different synthetic datasets with varying model and sample sizes to explore the impact of complexity and data size on efficiency. The two smaller examples both involved 10,000 observations. Regarding the number of levels for the two categorical variables, the first example included 15×20 levels, and the second 50×60. The third and larger example involved 113×143 levels and we have two variations, one with 10,000 and one with 80,000 observations to approximate the characteristics of the real dataset of the case study explored in Chapter 9.

Analysis of the results revealed agreement between the estimated and the true coefficients of the synthetic datasets for both JAGS and TWIiS. The posterior MCMC diagnostics to assess mixing and convergence included visual evaluation of traceplots and autocorrelation plots that did not present any visible anomalies. Overlaying the posterior density plots produced from the results of both JAGS and TWIiS showed overlapping plots and suggested similarity in the posterior estimates. The only inaccuracy in the findings was identified for JAGS and the Horseshoe prior method. In this case, some interaction coefficients that were simulated to be zero, were assigned high values by the program, wrongly suggesting that they should be included in the model. The effective sample size per second (ESS/sec) was used as an efficiency measure to account not only for the quality of mixing but also for the actual runtime. In all cases TWIiS was more efficient than the equivalent JAGS code. The degree of efficiency improvement using TWIiS varied depending on the model and variable selection method. For the small example with 15×20 levels and the assumption that both the main and interaction effects were asymmetric using the variable selection indicator method, the performance improvement with TWIiS reached approximately nine times against JAGS whereas, it dropped to approximately three times for the larger model with 113×143 levels. Conclusions from this chapter continued with comparison between the two variable selection approaches. The Kuo and Mallick method appeared to be more efficient than the Horseshoe prior in all examples considered. The additional inaccuracy of the Horseshoe prior that identified some false positive interactions as important, renders the use of variable selection indicators a safer approach to the variable selection problem considered. Comparison of these examples with escalated complexity helped us decide that using TWIiS with variable selection indicators is a better option for our real-world scientific application.

A secondary analysis in this chapter involved application of the Horseshoe prior along with Hamiltonian Monte Carlo through Stan in R and Rainier in Scala. The latter showed poorer performance, required longer runtimes and was less efficient than its counterparts. Regarding Stan, after exploring and identifying the appropriate tuning parameters, it showed fast mixing, and convergence in a small number of iterations, which counterbalanced the increased runtimes and led to efficiency metrics that showed similar performance to JAGS for the particular example that we studied. We also concluded that Stan showed faster mixing for the main effects, as opposed to JAGS and TWIiS that presented faster mixing for the interaction coefficients, which indicates that this difference could be deriving from using HMC or Gibbs sampler. Overall, the Gibbs sampler appeared to be more appropriate for our models.

In Chapter 8 we set the biological background and explained the miniQFA genetic experiment. We discussed its relevance to yeast genomics and telomere biology and its aim to identify combinations of genes, deletion of which has a potential impact on telomere capping defects that are caused by mutations on a particular non-essential gene for the survival of yeast strains. Various models could be explored regarding specific experimental characteristics of interest such as temperature and fitness measure. The results of fitting the models discussed in previous chapters, on some subsets of the yeast genome dataset were presented in Chapter 9. All subsets involved 111 different genes for the first gene deletion, that constitutes the first categorical variable, and 141 for the second, and a sample size of 83,000 observations. Regarding the programming implementation used to fit these models, we selected the parallel version of TWIiS using indicators for variable selection, since as we proved in earlier chapters this combination was the most efficient among the alternatives examined for the size and complexity of the model. Biological investigation is required to further validate the results.

10.3 Further work

In this section we will highlight some areas in which further work would be valuable. In this thesis we focused on using a Gibbs sampler for Bayesian variable selection on the interactions of two-way Anova models. Our main comparison was between TWIiS and JAGS and we only explored Stan and Rainier as alternative approaches. Stan was initially slow, but after tweaking its parameters it showed improvement. Further investigation could potentially decrease long runtimes that occurred from fitting more complex models.

From a statistical and computational perspective, another MCMC algorithm that we could explore is Reversible Jump MCMC (RJMCMC). This algorithm allows changes in the dimension of the parameter space between iterations. This characteristic appears in variable selection for which we observe a varying parameter space at every iteration depending on the variables that are included or excluded. It would be interesting to see the application of RJMCMC on the two-way Anova models with the specifications considered, and assess its results and performance. NIMBLE (2020) is an R package that has an RJMCMC sampler for variable selection and could be used for future exploration.

An alternative approach to MCMC algorithms, are the variational Bayesian inference methods, such as the mean field variational Bayes (MFVB). Rather than using sampling, variational inference is based on finding the best approximation of a distribution among a parameterised family using optimisation that minimises the Kullback-Leibler (KL) divergence to the exact posterior. Variational inference appears to be more computationally efficient than MCMC methods since it takes advantage of fast optimisation techniques and is often used for large datasets related to computational biology, computer vision and many others. However, variational inference results to a density that is close to the target, unlike MCMC methods that produce asymptotically exact samples from the target density (Blei *et al.*, 2018). Tan and Nott (2013) focused on applying variational Bayes methods for generalised linear mixed models (GLMMs) and tested centred, non-centred and partially non-centred parameterisations on various datasets. They concluded that the partially non-centred parameterisation produced a fit that was close to the result of a Gibbs sampler and had faster convergence. Therefore, variational inference techniques could potentially also be explored for our models in the future, even though it is not clear without further investigation whether it will be straightforward and beneficial to develop MFVB approximations for the non-standard models considered in this thesis.

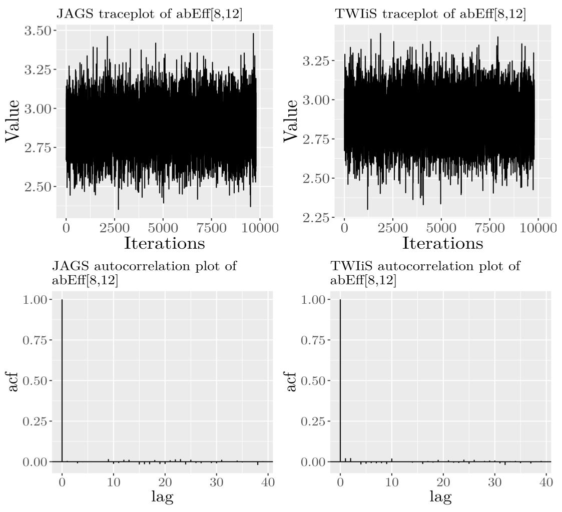
Regarding the code developed in Scala TWIiS is not a generic program at the moment. It is focused on specific modelling assumptions, for example regarding prior distributions, rendering model exploration less agile. Developing a probabilistic programming language and a corresponding library in Scala, could fill the current gap and facilitate Bayesian learning in the field of functional programming. Furthermore, we implemented a single chain sampling method. It would be interesting to see how the code could extend to embed and support multiple chains, and how we could take advantage of parallel programming and multiple cores to run these chains in parallel. This improvement could potentially lead to additional efficiency gain.

Concerning the application area and the yeast genome case study, further validation of the results, through experimental exploration and literature research, is necessary to assess which identified gene interactions could have scientific interest. Exploration of additional models and different distributions could reveal whether alternative models would be more appropriate in some cases. According to Heydari (2014), apart from the normal distribution used to model the orfΔ fitness, Student's t-distribution could also be appropriate, or even preferable. Finally, the model comparison applied in Chapter 9 was fairly informal, and the reason is twofold. Firstly, evaluation of biological models cannot be conducted strictly by using statistical criteria, as there are specific genetic behaviours that are expected to be observed in the results, and models that reflect these expectations are preferred by biologists. Secondly, Bayesian model comparison is a difficult statistical problem because according to Gelman *et al.* (2013), the first category of comparison techniques that are based on expected predictive accuracy, present flaws and are often unreliable. In a more fully Bayesian approach, the Bayes factor also presents challenges, as it involves the marginal likelihood that is sensitive to model assumptions. Therefore, further scientific judgement is necessary and decision upon model choice varies with context.

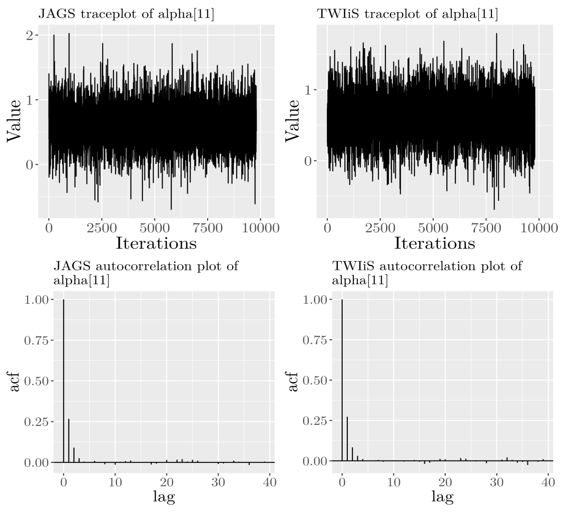
Appendix A

Variable selection using Horseshoe priors

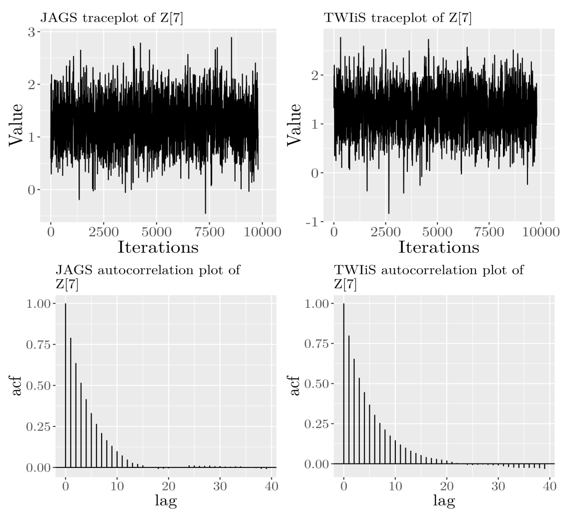
A.1 Synthetic dataset 15x20



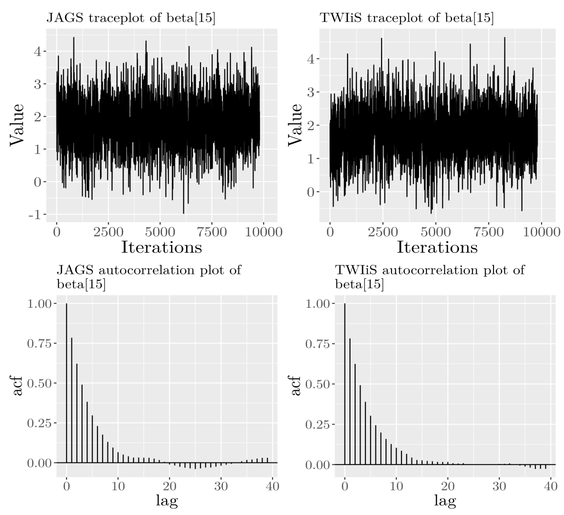
(a) Both symmetric - variable: $\gamma_{8,12}$



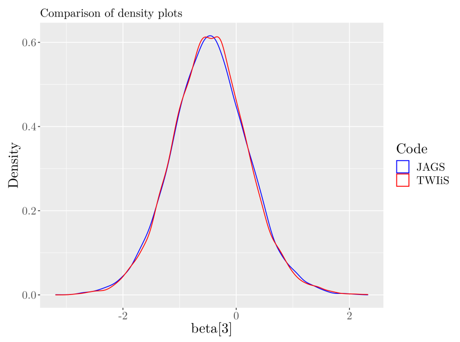
(b) Symmetric interactions - variable: α_{11}



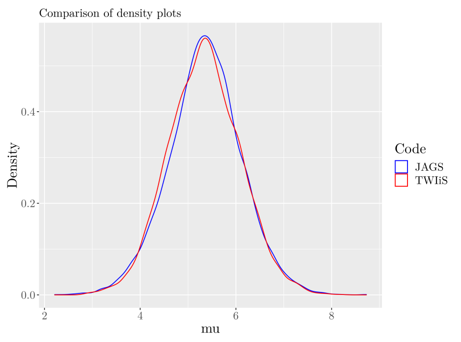
(c) Symmetric main - variable: Z_7



(d) Both asymmetric - variable: β_{15}

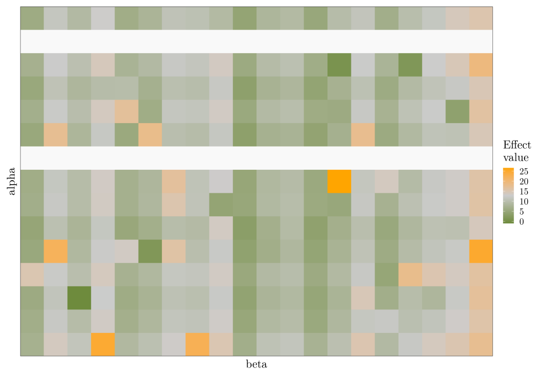


(e) Both asymmetric - variable: β_3

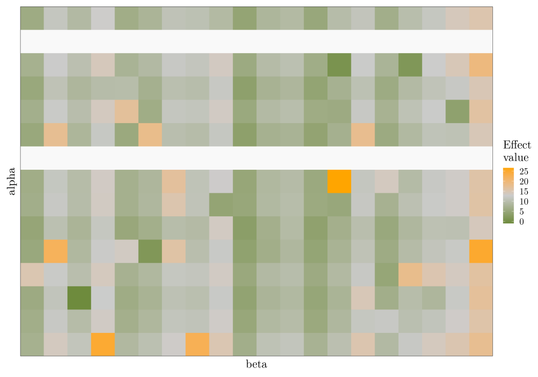


(f) Both symmetric - variable: μ

Figure A.1: Diagnostic plots (a, b, c, d) and posterior density plots comparison (e, f) for variable selection with Horseshoe prior. Example 1: 10×15 levels.



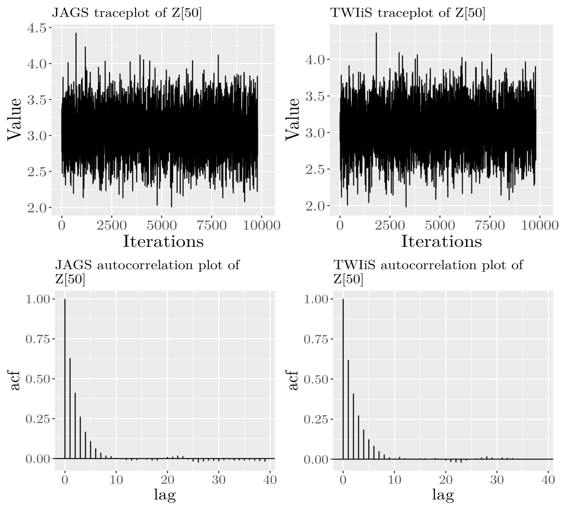
(a) JAGS



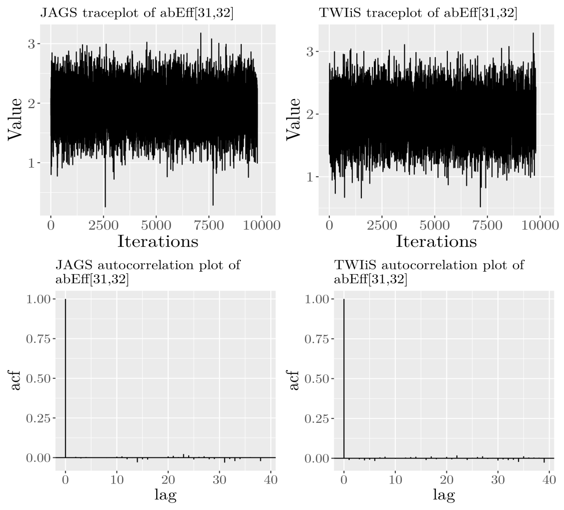
(b) TWIiS

Figure A.2: Heatmaps of the interaction strengths using variable selection with the Horseshoe prior for the case of both main and interaction effects as asymmetric. Example 1: 15×20

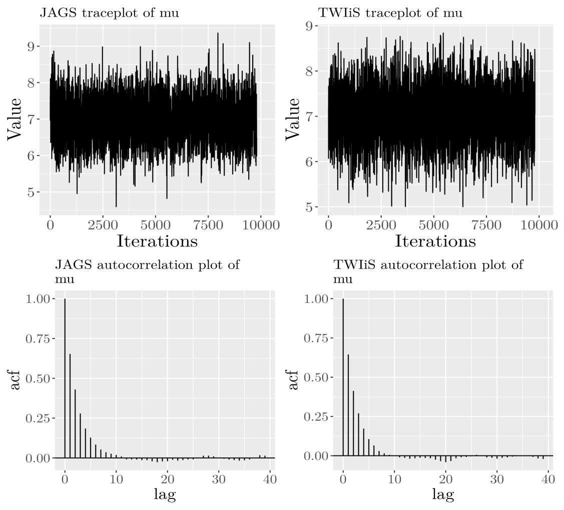
A.2 Synthetic dataset 50x60



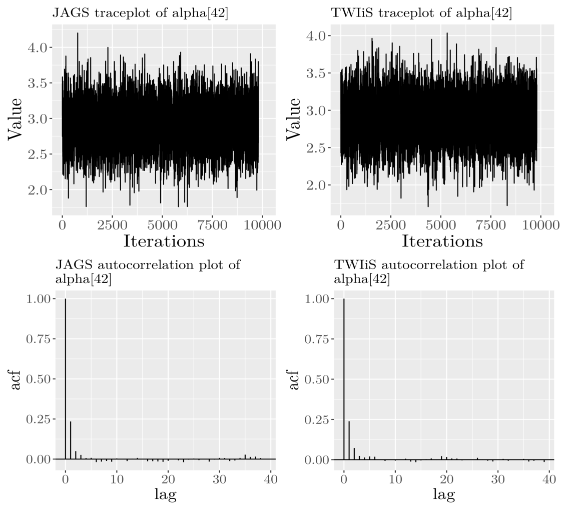
(a) Both symmetric - variable: Z_{50}



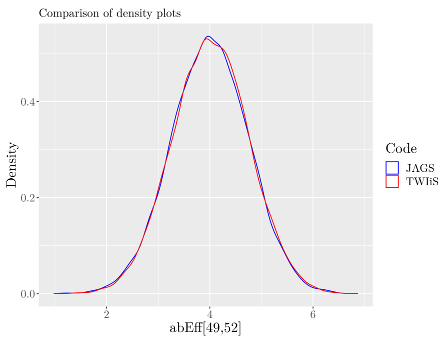
(b) Symmetric interactions - variable: $\gamma_{31,32}$



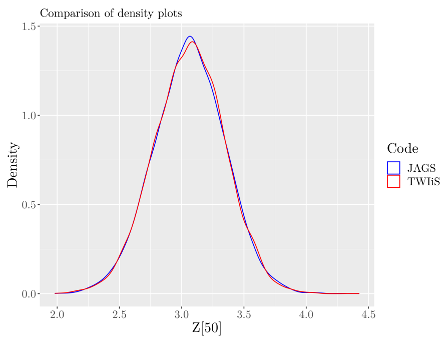
(c) Symmetric main - variable: μ



(d) Both asymmetric - variable: α_{42}



(e) Both asymmetric - variable: $\gamma_{49,52}$



(f) Both symmetric - variable: Z_{50}

Figure A.3: Diagnostic plots (a, b, c, d) and posterior density plots comparison (e, f) for variable selection with Horseshoe priors. Example 2: 50×60 levels.

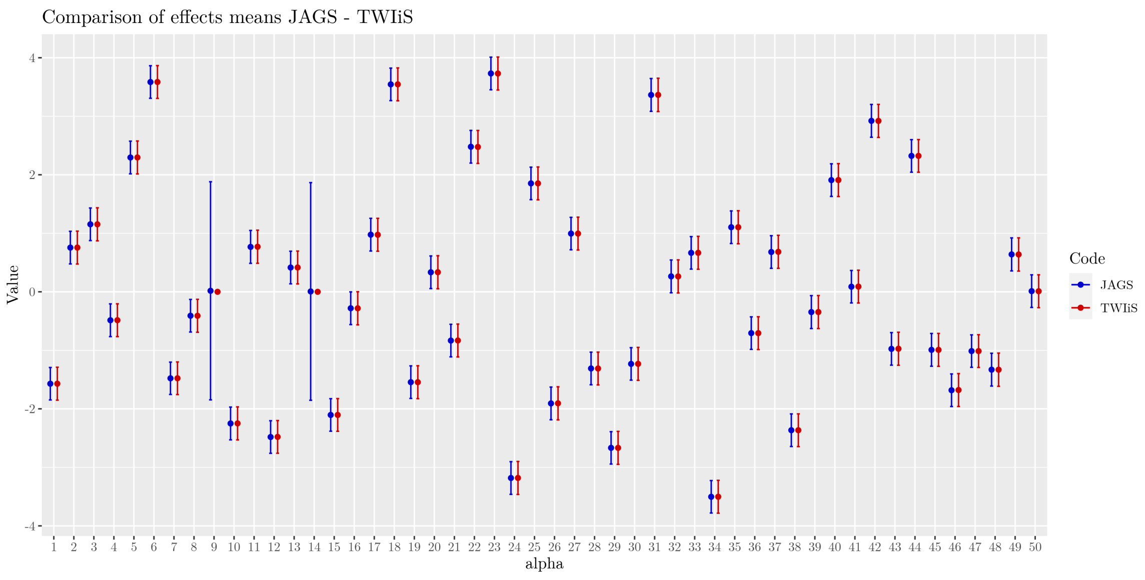
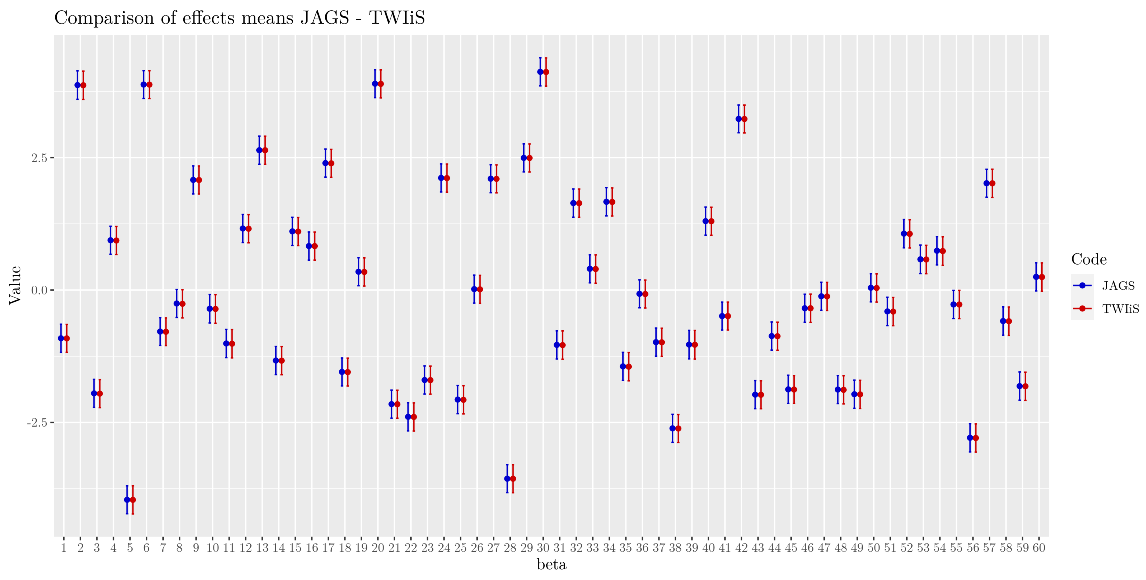
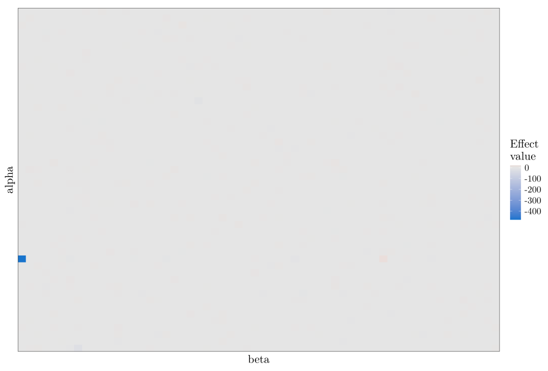


Figure A.4: (a) Comparison - αs

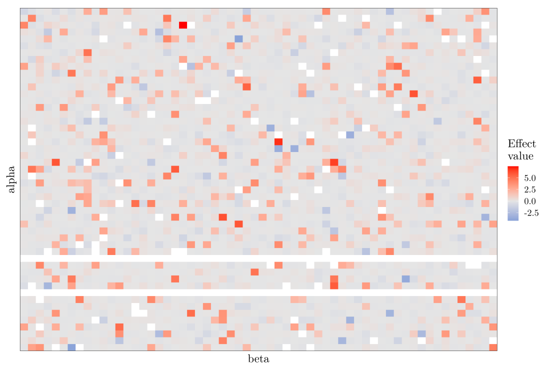


(b) Comparison - βs

Figure A.4: Comparison of main effects means estimated from JAGS and TWIiS for the case of symmetric interactions. Example 2: 50×60

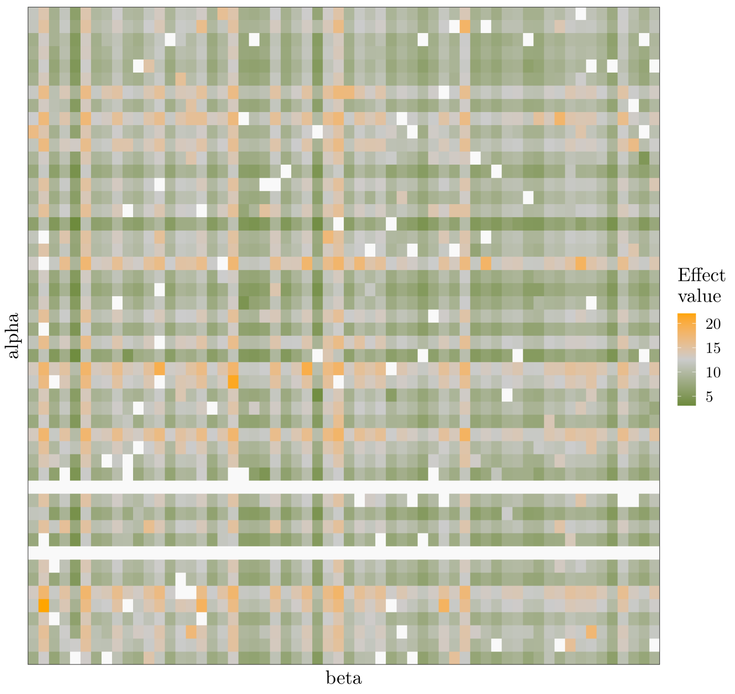


(a) JAGS

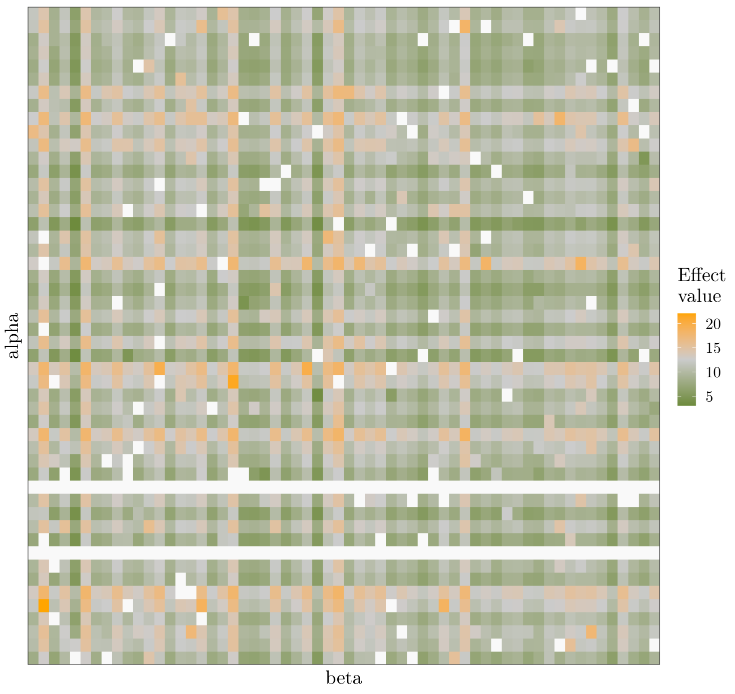


(b) TWIiS

Figure A.5: Heatmaps of the interaction strengths using variable selection with the Horseshoe prior for the case of both main and interaction effects as asymmetric. Plot (a) highlights problematic identification $\gamma_{14,1} = -600$ for JAGS. Removal of this value leads to a JAGS heatmap similar to TWIiS. Example 2: 50×60



(a) JAGS



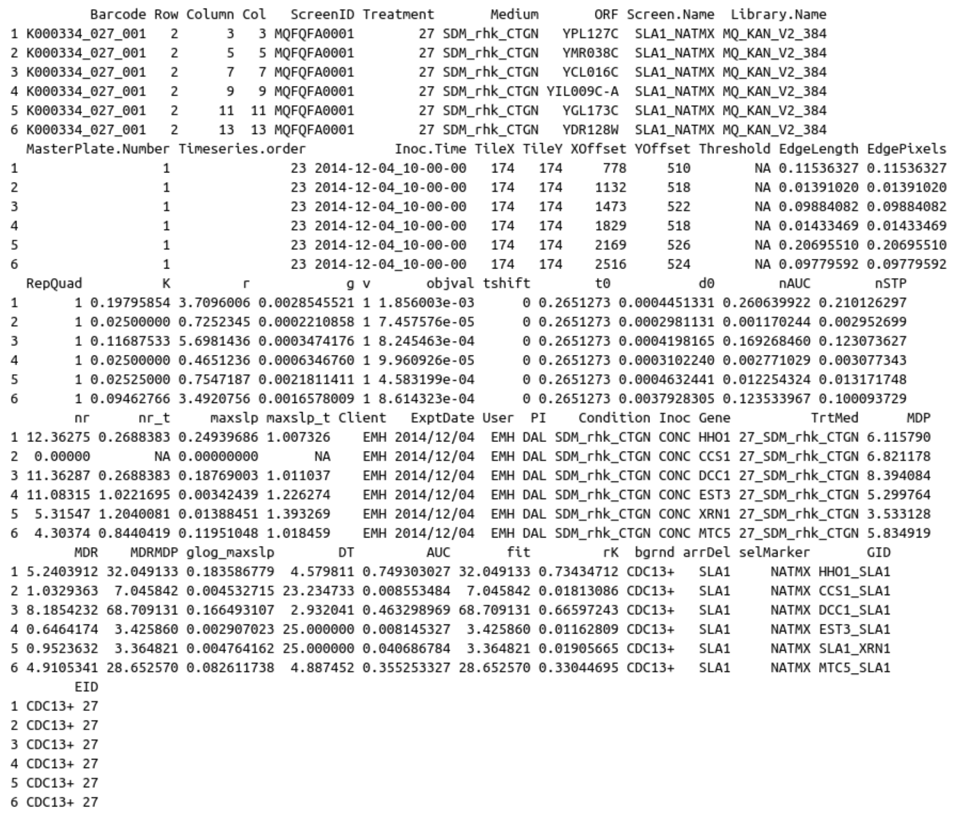
(b) TWIiS

Figure A.6: Heatmaps of the group means using variable selection with the Horseshoe prior for the case of symmetric interactions. Example 2: 50×60

Appendix B

Mini QFA

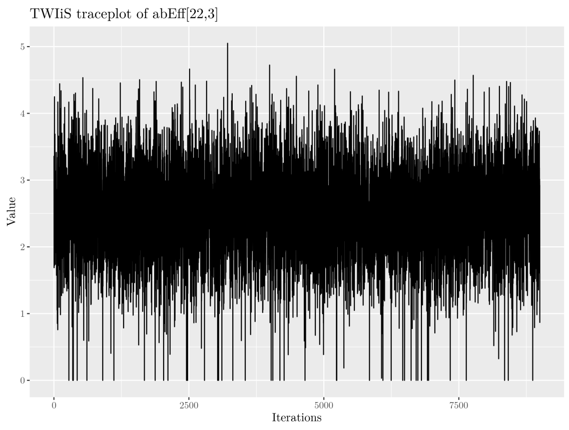
B.1 mini QFA dataset sample



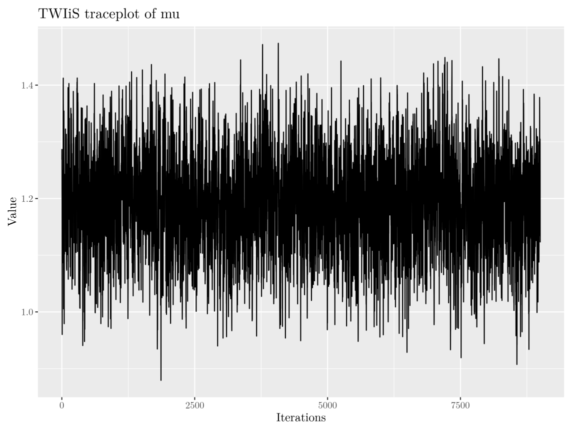
```
         Barcode Row Column Col   ScreenID Treatment       Medium       ORF Screen.Name  Library.Name
1 K000334_027_001   2      3   3 MQFQFA0001        27 SDM_rhk_CTGN   YPL127C  SLA1_NATMX MQ_KAN_V2_384
2 K000334_027_001   2      5   5 MQFQFA0001        27 SDM_rhk_CTGN   YMR038C  SLA1_NATMX MQ_KAN_V2_384
3 K000334_027_001   2      7   7 MQFQFA0001        27 SDM_rhk_CTGN   YCL016C  SLA1_NATMX MQ_KAN_V2_384
4 K000334_027_001   2      9   9 MQFQFA0001        27 SDM_rhk_CTGN YIL009C-A  SLA1_NATMX MQ_KAN_V2_384
5 K000334_027_001   2     11  11 MQFQFA0001        27 SDM_rhk_CTGN   YGL173C  SLA1_NATMX MQ_KAN_V2_384
6 K000334_027_001   2     13  13 MQFQFA0001        27 SDM_rhk_CTGN   YDR128W  SLA1_NATMX MQ_KAN_V2_384
  MasterPlate.Number Timeseries.order           Inoc.Time TileX TileY XOffset YOffset Threshold EdgeLength EdgePixels
1                  1               23 2014-12-04_10-00-00   174   174     778     510        NA 0.11536327 0.11536327
2                  1               23 2014-12-04_10-00-00   174   174    1132     518        NA 0.01391020 0.01391020
3                  1               23 2014-12-04_10-00-00   174   174    1473     522        NA 0.09884082 0.09884082
4                  1               23 2014-12-04_10-00-00   174   174    1829     518        NA 0.01433469 0.01433469
5                  1               23 2014-12-04_10-00-00   174   174    2169     526        NA 0.20695510 0.20695510
6                  1               23 2014-12-04_10-00-00   174   174    2516     524        NA 0.09779592 0.09779592
  RepQuad          K         r            g v       objval tshift        t0           d0        nAUC        nSTP
1       1 0.19795854 3.7096006 0.0028545521 1 1.856003e-03      0 0.2651273 0.0004451331 0.260639922 0.210126297
2       1 0.02500000 0.7252345 0.0002210858 1 7.457576e-05      0 0.2651273 0.0002981131 0.001170244 0.002952699
3       1 0.11687533 5.6981436 0.0003474176 1 8.245463e-04      0 0.2651273 0.0004198165 0.169268460 0.123073627
4       1 0.02500000 0.4651236 0.0006346760 1 9.960926e-05      0 0.2651273 0.0003102240 0.002771029 0.003077343
5       1 0.02525000 0.7547187 0.0021811411 1 4.583199e-04      0 0.2651273 0.0004632441 0.012254324 0.013171748
6       1 0.09462766 3.4920756 0.0016578009 1 8.614323e-04      0 0.2651273 0.0037928305 0.123533967 0.100093729
        nr      nr_t     maxslp maxslp_t Client   ExptDate User  PI    Condition Inoc Gene          TrtMed      MDP
1 12.36275 0.2688383 0.24939686 1.007326    EMH 2014/12/04  EMH DAL SDM_rhk_CTGN CONC HHO1 27_SDM_rhk_CTGN 6.115790
2  0.00000        NA 0.00000000       NA    EMH 2014/12/04  EMH DAL SDM_rhk_CTGN CONC CCS1 27_SDM_rhk_CTGN 6.821178
3 11.36287 0.2688383 0.18769003 1.011037    EMH 2014/12/04  EMH DAL SDM_rhk_CTGN CONC DCC1 27_SDM_rhk_CTGN 8.394084
4 11.08315 1.0221695 0.00342439 1.226274    EMH 2014/12/04  EMH DAL SDM_rhk_CTGN CONC EST3 27_SDM_rhk_CTGN 5.299764
5  5.31547 1.2040081 0.01388451 1.393269    EMH 2014/12/04  EMH DAL SDM_rhk_CTGN CONC XRN1 27_SDM_rhk_CTGN 3.533128
6  4.30374 0.8440419 0.11951048 1.018459    EMH 2014/12/04  EMH DAL SDM_rhk_CTGN CONC MTC5 27_SDM_rhk_CTGN 5.834919
        MDR    MDRMDP glog_maxslp        DT         AUC       fit         rK  bgrnd arrDel selMarker       GID
1 5.2403912 32.049133 0.183586779  4.579811 0.749303027 32.049133 0.73434712 CDC13+   SLA1     NATMX HHO1_SLA1
2 1.0329363  7.045842 0.004532715 23.234733 0.008553484  7.045842 0.01813086 CDC13+   SLA1     NATMX CCS1_SLA1
3 8.1854232 68.709131 0.166493107  2.932041 0.463298969 68.709131 0.66597243 CDC13+   SLA1     NATMX DCC1_SLA1
4 0.6464174  3.425860 0.002907023 25.000000 0.008145327  3.425860 0.01162809 CDC13+   SLA1     NATMX EST3_SLA1
5 0.9523632  3.364821 0.004764162 25.000000 0.040686784  3.364821 0.01905665 CDC13+   SLA1     NATMX SLA1_XRN1
6 4.9105341 28.652570 0.082611738  4.887452 0.355253327 28.652570 0.33044695 CDC13+   SLA1     NATMX MTC5_SLA1
        EID
1 CDC13+ 27
2 CDC13+ 27
3 CDC13+ 27
4 CDC13+ 27
5 CDC13+ 27
6 CDC13+ 27
```

Figure B.1: miniQFA dataset sample. Notable columns used for modelling are: "r", "K" and "rK" used as fitness measures, "EID" that shows the background screen and the temperature, and "GID" that shows the two "deleted" genes.

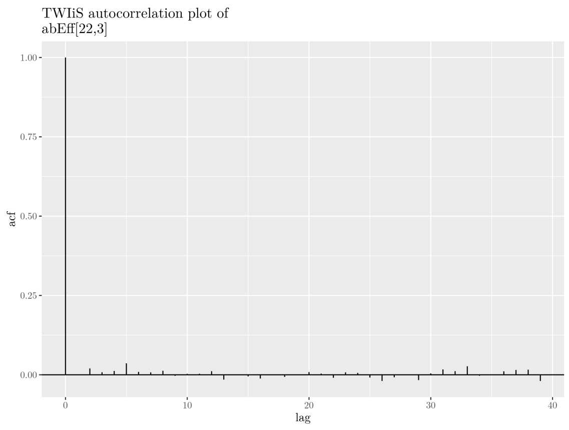
B.2 Statistical modelling



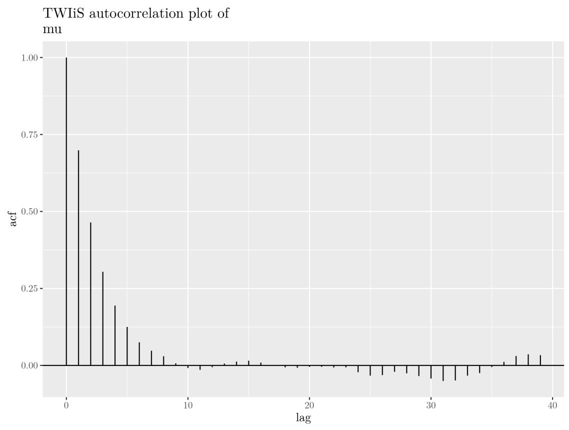
(a) Traceplot - effect $\gamma_{22,3}$



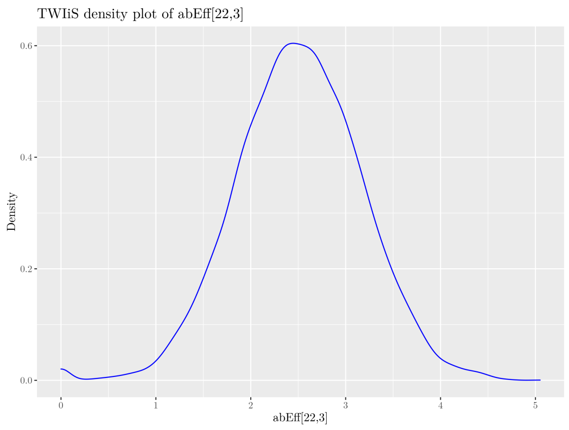
(b) Traceplot - μ



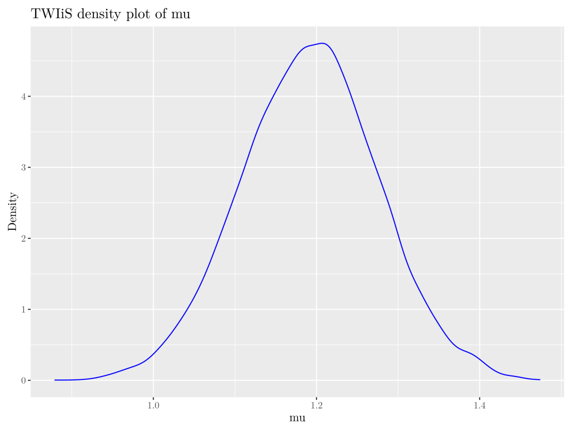
(c) Autocorrelation plot - effect $\gamma_{22,3}$



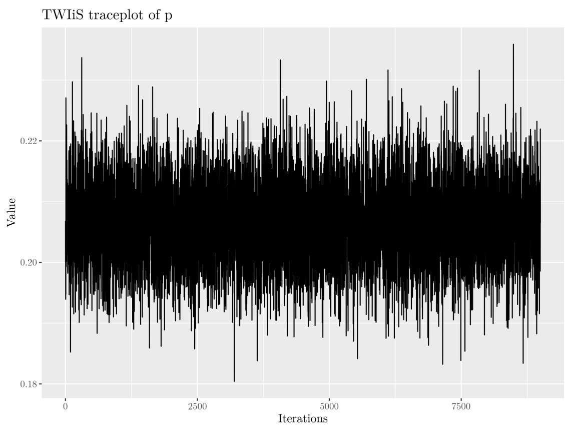
(d) Autocorrelation plot - μ



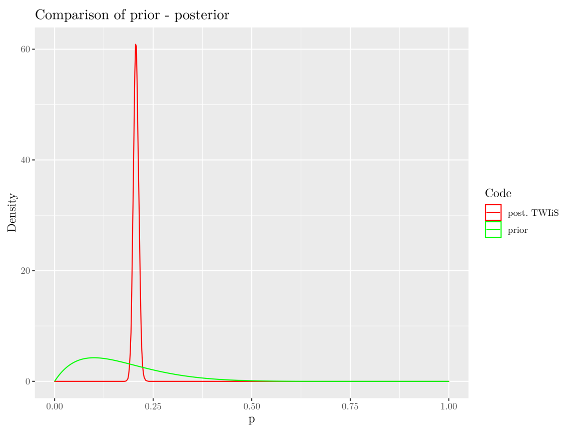
(e) Density plot - effect $\gamma_{22,3}$



(f) Density plot - μ

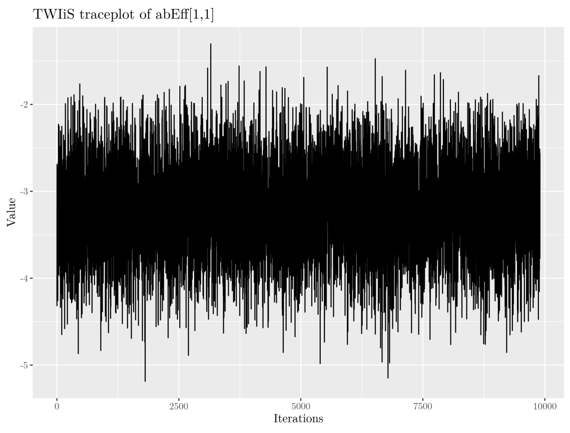


(g) Traceplot - inclusion probability p

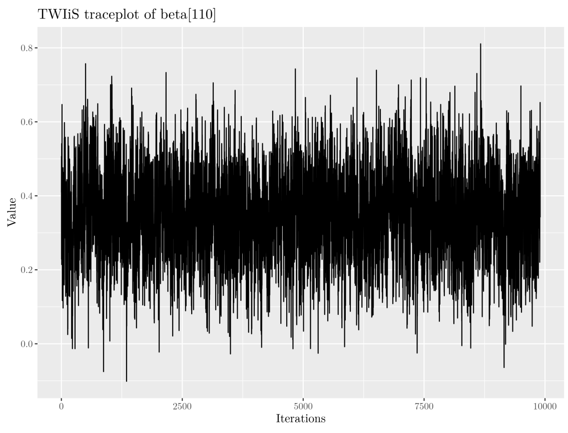


(h) Prior-posterior density plot - p

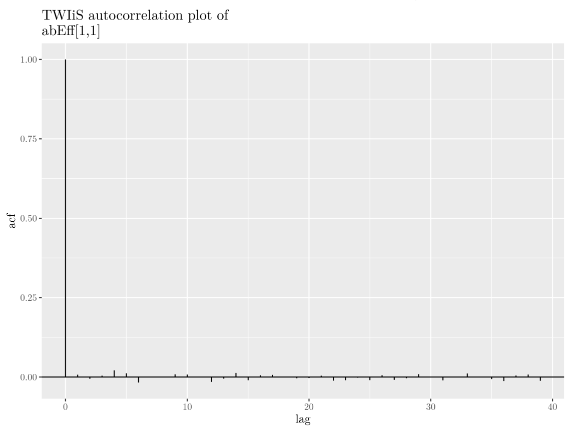
Figure B.2: Traceplots, autocorrelation and density plots for the experiment: *cdc13-1* at 33°C with r as measure of fitness. Both the main and interaction effects are asymmetric.



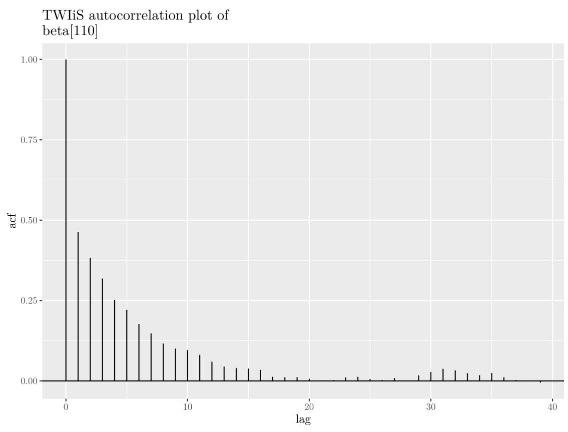
(a) Traceplot - effect $\gamma_{1,1}$



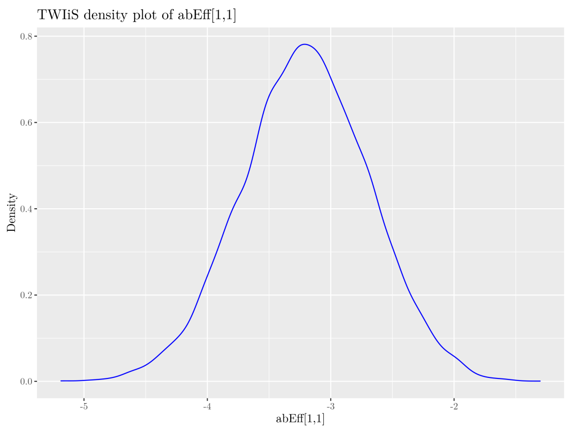
(b) Traceplot - β_{110}



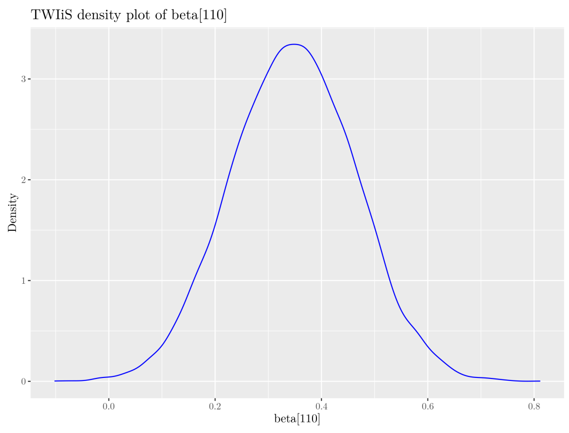
(c) Autocorrelation plot - effect $\gamma_{1,1}$



(d) Autocorrelation plot - β_{110}

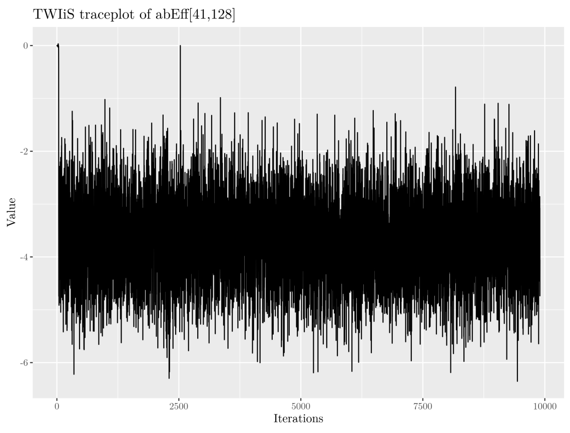


(e) Density plot - effect $\gamma_{1,1}$

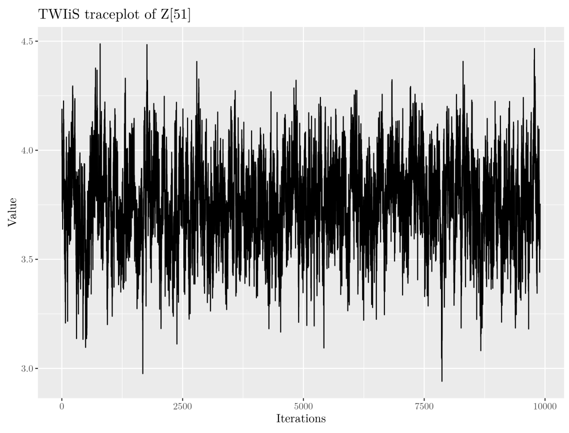


(f) Density plot - β_{110}

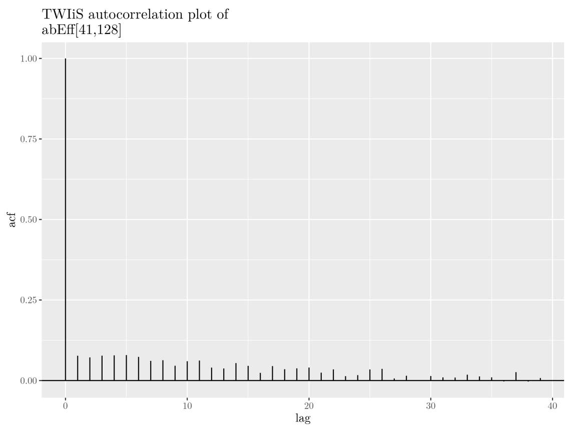
Figure B.3: Traceplots, autocorrelation and density plots for the experiment: *cdc13-1* at 27°C with r as measure of fitness. Both the main and interaction effects are symmetric.



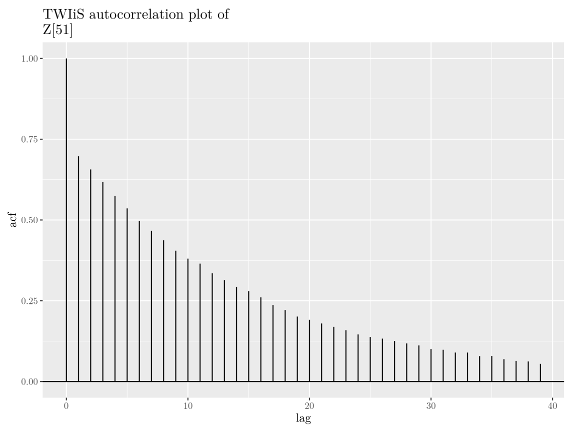
(a) Traceplot - effect $\gamma_{41,128}$



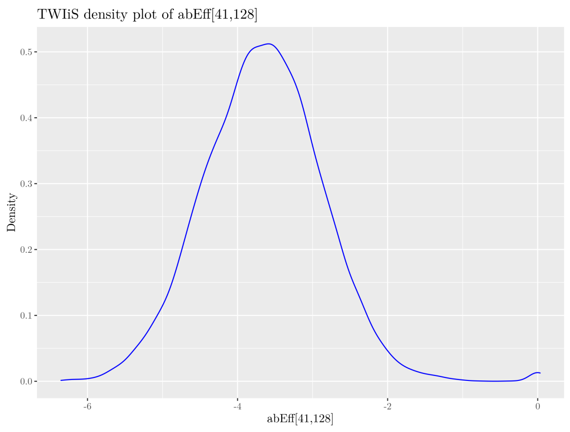
(b) Traceplot - Z_{51}



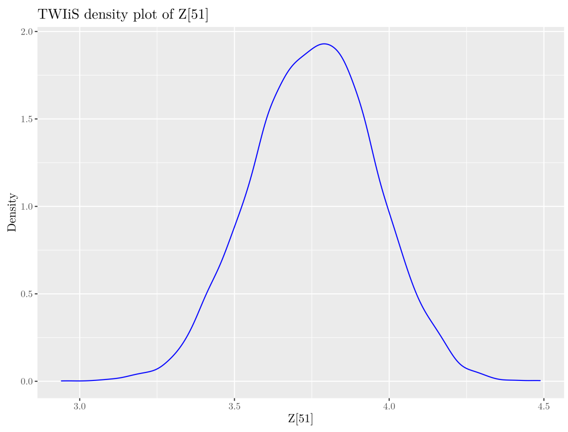
(c) Autocorrelation plot - effect $\gamma_{41,128}$



(d) Autocorrelation plot - Z_{51}



(e) Density plot - effect $\gamma_{41,128}$



(f) Density plot - Z_{51}

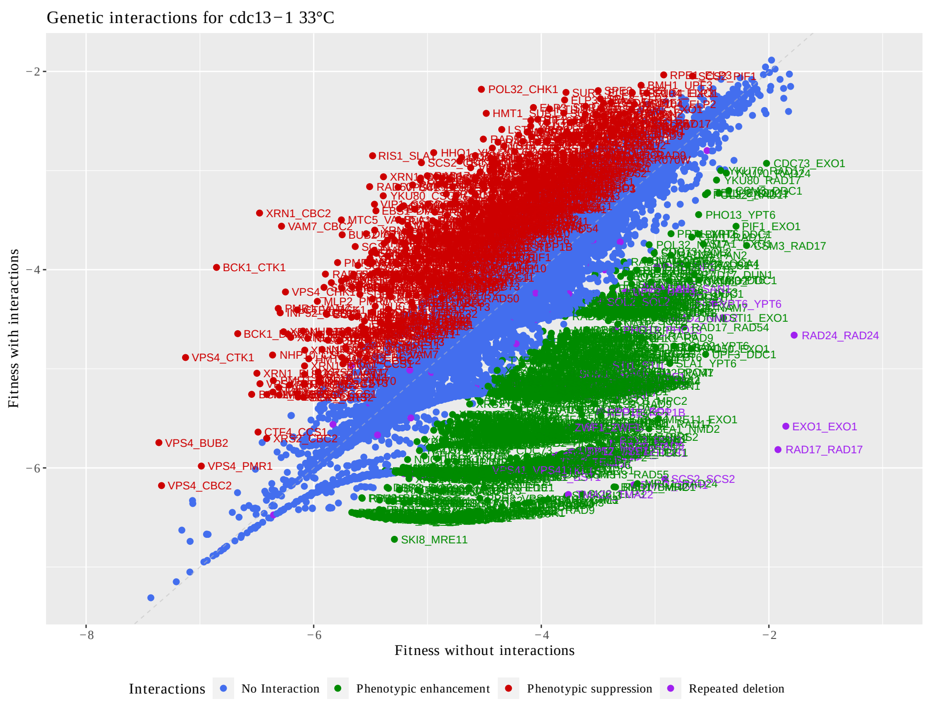
Figure B.4: Traceplots, autocorrelation and density plots for the experiment: *CDC13+* at 33°C with r as measure of fitness. Both the main and interaction effects are asymmetric.

Table B.1: Top 20 enhancer combinations, ranked by strength of estimated interaction for the experiment: *cdc13-1* at 33°C with r as measure of fitness. Cases: "Both asymmetric", "Both symmetric" and "Symmetric interactions".

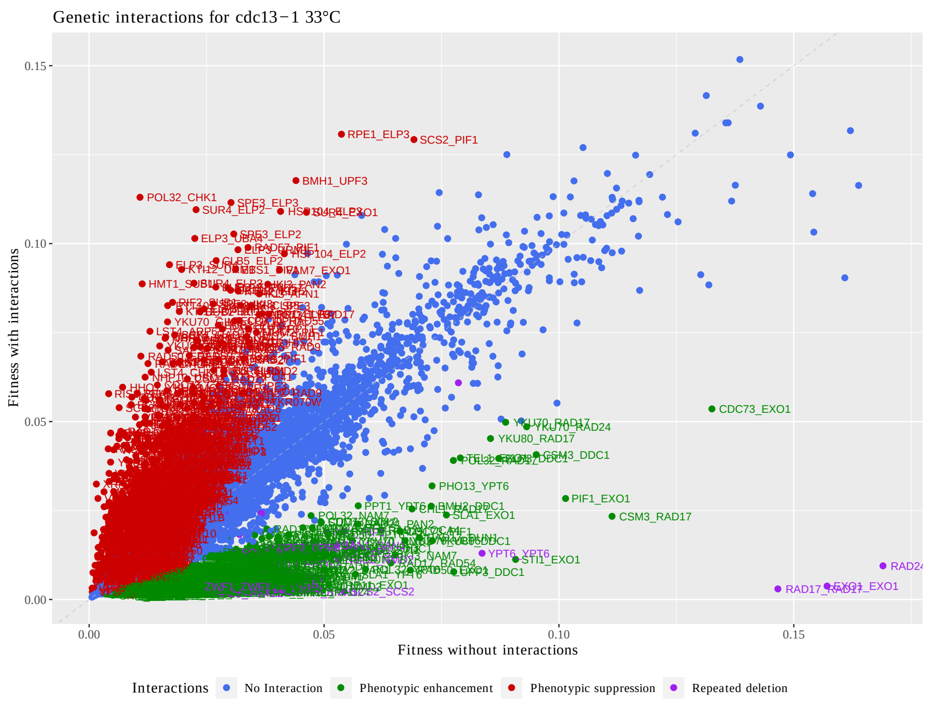
Both asymmetric		Both symmetric		Symmetric interactions	
Genes	Strength	Genes	Strength	Genes	Strength
RAD9_DUN1	-4.76	DUN1_RAD17	-4.63	MRC1_RAD17	-4.15
MRC1_RAD24	-4.73	MRC1_RAD24	-4.07	MRC1_RAD24	-4.15
MRC1_RAD17	-4.53	DDC1_MRC1	-4.01	RAD9_DUN1	-4.15
CHK1_RAD9	-4.34	MRC1_RAD17	-3.88	MRC1_DDC1	-4.12
MRC1_DDC1	-4.09	DUN1_RAD9	-3.86	UPF3_DDC1	-3.81
UPF3_DDC1	-3.94	CSM3_RAD24	-3.78	RAD17_DUN1	-3.73
RAD17_DUN1	-3.76	DDC1_UPF3	-3.70	RAD24_CSM3	-3.65
RAD9_SCS2	-3.72	CTF18_RAD24	-3.59	POL32_RAD24	-3.62
RAD9_CSM3	-3.66	POL32_RAD24	-3.55	YKU80_DDC1	-3.57
RAD9_UPF3	-3.65	CBC2_RAD24	-3.55	CHK1_RAD9	-3.46
YKU80_DDC1	-3.65	DDC1_YKU80	-3.50	SCS2_DDC1	-3.44
RAD9_MRC1	-3.61	DIA2_RAD24	-3.43	DIA2_RAD24	-3.33
POL32_RAD24	-3.60	CSM3_RAD17	-3.39	CSM3_RAD17	-3.23
RAD9_RAD54	-3.59	CTF18_RAD17	-3.33	VPS4_DDC1	-3.19
DIA2_RAD17	-3.59	CHK1_RAD9	-3.33	MRC1_RAD9	-3.19
CSM3_RAD17	-3.57	PMR1_RAD24	-3.32	PMR1_RAD24	-3.18
DIA2_RAD24	-3.56	MRC1_RAD9	-3.31	DIA2_RAD17	-3.17
SCS2_DDC1	-3.49	DDC1_VPS4	-3.28	RAD24_CTF18	-3.14
RAD9_RAD27	-3.49	CTK1_RAD17	-3.25	CSM3_DDC1	-3.13
CSM3_RAD24	-3.47	DDC1_SCS2	-3.23	XRS2_DDC1	-3.11

Table B.2: Top 20 suppressor combinations, ranked by strength of estimated interaction for the experiment: *cdc13-1* at 33°C with r as measure of fitness. Cases: "Both asymmetric", "Both symmetric" and "Symmetric interactions".

Both asymmetric		Both symmetric		Symmetric interactions	
Genes	Strength	Genes	Strength	Genes	Strength
MBP1_BAS1	4.67	BAS1_OCA4	3.78	BMH2_RAD9	3.64
RAD9_FKH2	4.68	MAD1_RAD9	3.83	FKH2_RAD9	3.69
RAD9_BCK1	4.81	CHK1_MAD1	3.86	EBS1_MAD1	3.69
HGH1_RAD9	4.82	DPH1_DUN1	3.87	HHO1_RAD9	3.73
OCA4_RAD9	4.87	FKH1_RAD9	3.88	RAD17_SLA1	3.76
CHK1_RIF2	4.89	CHK1_EDE1	3.90	DPH1_DUN1	3.89
SUM1_RAD9	4.90	HGH1_RAD9	3.93	FKH1_RAD9	3.93
CHK1_PAN2	4.93	DDC1_SLA1	4.10	MBP1_BAS1	3.98
RAD9_RIF2	5.02	DDC1_MLP2	4.11	RAD9_YKR070W	4.01
RAD9_HHO1	5.17	RAD9_YKR070W	4.37	HGH1_RAD9	4.06
RAD17_BCK1	5.25	MLP2_RAD9	4.43	MLP2_DDC1	4.15
SOL2_RAD9	5.29	BCK1_RAD9	4.43	BCK1_RAD9	4.47
PAN2_DDC1	5.33	RAD9_SOL2	4.58	MLP2_RAD9	4.48
PAN2_RAD17	5.36	BAS1_MBP1	4.66	RAD9_SOL2	4.62
PAN2_RAD9	5.51	BCK1_RAD17	4.81	RAD9_VPS74	4.64
MLP2_RAD9	5.78	DDC1_PAN2	4.81	BCK1_RAD17	4.82
PAN2_RAD24	5.93	RAD9_VPS74	4.98	PAN2_DDC1	4.89
PPT1_RAD9	6.39	PAN2_RAD9	5.11	PAN2_RAD9	5.17
RAD9_EDE1	6.49	PPT1_RAD9	5.60	PPT1_RAD9	5.75
MBP1_RAD9	6.66	EDE1_RAD9	6.87	RAD9_EDE1	6.51



(a) Logarithmic scale



(b) Natural scale

Figure B.5: Genetic interactions plot for the experiment: *cdc13-1* at 33°C with $\log K$ as measure of fitness. Both the main and interaction effects are asymmetric. (a) Logarithmic scale. Patterns suggest potential non-linearity. (b) Natural scale.

Bibliography

Adam, N., Degelman, E., Briggs, S., Wazen, R., Colarusso, P., Riabowol, K. and Beattie, T. (2019) Telomere analysis using 3D fluorescence microscopy suggests mammalian telomere clustering in hTERT-immortalized Hs68 fibroblasts. *Communications Biology* **2**.

Addinall, S., Holstein, E.-M., Lawless, C., Yu, M., Chapman, K. and Banks, A. (2011) Quantitative Fitness Analysis Shows That NMD Proteins and Many Other Protein Complexes Suppress or Enhance Distinct Telomere Cap Defects. *PLoS Genet* **7** (4), doi:10.1371/journal.pgen.1001362.

Al-hayanni, M. A. N., Xia, F., Rafiev, A., Romanovsky, A., Shafik, R. and Yakovlev, A. (2020) Amdahl's law in the context of heterogeneous many-core systems - A survey. *IET Computers & Digital Techniques* **14** (4), 133–148.

Alexander, A. (2014) *Functional Programming, Simplified: (Scala Edition)*. CreateSpace Independent Publishing Platform.

Angelino, E., Johnson, M. and Adams, R. P. (2016) Patterns of Scalable Bayesian Inference. *arXiv:1602.05221v2 [stat.ML]* .

Aubert, G. and Lansdorp, P. M. (2008) Telomeres and Aging. *Physiological Reviews* **88** (2), 557–579.

Avery, L. and Wasserman, S. (1992) Ordering gene function: the interpretation of epistasis in regulatory hierarchies. *Trends Genet* **8** (9), 312–316.

Banks, A. P., Lawless, C. and Lydall, D. A. (2012) A Quantitative Fitness Analysis Workflow. *Journal of Visualized Experiments* **66**, doi:10.3791/4018.

Banterle, M., Grazian, C. and Robert, P. (2014) Accelerating Metropolis-Hastings algorithms: Delayed acceptance with prefetching. *arXiv:1406.2660 [stat.CO]* .

Bardenet, R., Doucet, A. and Holmes, C. (2014) Towards scaling up Markov Chain Monte Carlo: an adaptive subsampling approach. *ICML* pp. 405–413.

Betancourt, M. (2018*a*) A Conceptual Introduction to Hamiltonian Monte Carlo. *arXiv:1701.02434v2[stat.ME]* .

Betancourt, M. (2018*b*) Fitting the Cauchy distribution. https://betanalpha.github.io/assets/case_studies/fitting_the_cauchy.html, accessed August 2020.

Blei, D. M., Kucukelbir, A. and McAuliffe, J. D. (2018) Variational Inference: A Review for Statisticians. *arXiv:1601.00670v9 [stat.CO]* .

Boocock, D. and Lawless, C. (2017) Modelling competition for nutrients between microbial populations growing on a solid agar surface. *bioRxiv* Doi:10.1101/086835.

Breeze (2020) Library for numerical processing. https://github.com/scalanlp/breeze, accessed August 2020.

Brooks, S. P. and Roberts, G. O. (1998) Convergence assessment techniques for Markov chain Monte Carlo. *Statistics and Computing* **8**, 319–335.

Bryant, A. (2018) Rainier. https://github.com/stripe/rainier, accessed August 2020.

Bryant, A. (2020) Rainier half-Cauchy. https://github.com/stripe/rainier/issues/478, accessed August 2020.

BUGS (1989) The BUGS project. https://www.mrc-bsu.cam.ac.uk/software/bugs/, accessed July 2020.

Bumbaca, F., Misra, S. and Rossi, E. (2017) Distributed Markov Chain Monte Carlo for Bayesian Hierarchical Models. Available at SSRN: https://ssrn.com/abstract=2964646 or http://dx.doi.org/10.2139/ssrn.2964646.

Carvalho, C. M., Polson, N. G. and Scott, J. (2008) The Horseshoe estimator for Sparse Signals. *Biometrica* **97** (2), 465–480.

Carvalho, C. M., Polson, N. G. and Scott, J. (2009) Handling Sparsity via the Horseshoe. *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics* **5**.

Cats (2019*a*) Cats - kleisli. https://typelevel.org/cats/datatypes/kleisli.html, accessed August 2020.

Cats (2019*b*) Cats - type classes. Https://typelevel.org/cats/typeclasses.html, accessed July 2020.

Chiusano, P. and Bjarnason, R. (2017) *Functional Programming in Scala*. Manning.

DasMurtey, M. and Ramasamy, P. (2016) Sample Preparations for scanning electron microscopy – Life sciences. *Modern Electron Microscopy in Physical and Life Sciences* pp. 161–185.

Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L. and White, A. (2003) *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers.

EMBL-EBI (2020) Checkpoint protein rad17/rad24. https://www.ebi.ac.uk/interpro/entry/InterPro/IPR004582/, accessed October 2020.

Erwig, M. and Kollmansberger, S. (2006) Probabilistic Functional Programming in Haskell. *Journal of Functional Programming* .

Friel, N., Mira, A. and Oates, C. (2016) Exploiting Multi-Core Architectures for Reduced-Variance Estimation with Intractable Likelihoods. *Bayesian Analysis* **11** (1), 215–245.

Gamerman, D. and Lopes, H. (2006) *Markov Chain Monte Carlo: stochastic simulation for Bayesian inference*. ed. 2nd Chapman and Hall/CRC.

Gelman, A., Carlin, B. J., Stern, S. H., Dunson, B. D., Vehtari, A. and Rubin, B. D. (2013) *Bayesian Data Analysis*. ed. 3rd Boca Raton: CRC Press/Taylor and Francis Group.

George, E. I. and McCulloch, R. E. (1993) Variable Selection via Gibbs Sampling. *Journal of the American Statistical Association* **88** (423), 881–889.

Geyer, C. J. (1991) Markov chain Monte Carlo maximum likelihood. *Proceedings of the 23rd Symposium on the Interface* Interface Foundation of North America, E.M. Keramidas (editor).

Giry, M. (1982) A categorical approach to probability theory. In Categorical aspects of topology and analysis. *Springer* pp. 68–85.

Gobbini, E., Cassani, C., Villa, M., Bonetti, D. and Longhese, P. (2016) Functions and regulation of the MRX complex at DNA double-strand breaks. *Microbial Cell* **3** (8), 329–337.

Gorham, J. and Mackey, L. (2015) Measuring sample quality with Stein's method. *NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems* **1**, 226–234.

Goudie, R. J. B., Turner, R. M., DeAngelis, D. and Thomas, A. (2018) MultiBUGS: A parallel implementation of the BUGS modelling framework for faster bayesian inference. *arXiv:1704.03216v4 [stat.CO]* .

Grozev, N. (2016) Functional Programming and Category Theory [Part 1] - Categories and Functors. https://nikgrozev.com/2016/03/14/functional-programming-and-category-theory-part-1-categories-and-functors/, accessed July 2020.

Gustaphson, J. L. (1988) Reevaluating Amdahl's law. *Communications of the ACM* **31** (5).

Hartl, D. L. and Jones, E. W. (1998) *Genetics: Principles and analysis*. 4th ed. Jones and Bartlett Publishers Inc.

Herrmann, M., Pusceddu, I., Mrz, W. and Herrmann, W. (2018) Telomere biology and age-related diseases. *Clinical Chemistry and Labolatory Medicine* **56** (8), 1210–1222.

Heydari, J. (2014) Bayesian Hierarchical Modelling for inferring genetic interactions in yeast. *arXiv:1405.7091v1 [stat.AP]* .

Heydari, J., Lawless, C., Lydall, D. and Wilkinson, D. (2016) Bayesian hierarchical modelling for inferring genetic interactions in yeast. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* **65** (3).

Hoff, P. D. (2009) *A First Course in Bayesian Statistical Methods*. Springer Publishing Company.

Hoffman, M. and Gelman, A. (2011) The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *arXiv:1111.4246v1 [stat.CO]* .

International Human Genome Sequencing Consortium (2004) Finishing the euchromatic sequence of the human genome. *Nature* **431**, 931–945, https://doi.org/10.1038/nature03001.

Jackman, S. (2009) *Bayesian Analysis for the Social Sciences*. Wiley.

Jelvis, T. (2017) The probability monad. Compose conference, https://www.youtube.com/watch?v=qZ4O-1VYv4c, accessed December 2019.

Johndrow, E., Orenstein, P. and Bhattacharya, A. (2018) Scalable MCMC for Bayes Shrinkage Priors. *arXiv:1705.00841v2 [stat.CO]* .

Johnson, V. E. and Rossell, D. (2012) Bayesian Model Selection in High-Dimensional Settings. *Journal of the American Statistical Association* pp. 649–660.

Korattikara, A., Chen, Y. and Welling, M. (2014) Austerity in MCMC land: Cutting the Metropolis-Hastings budget. *arXiv:1304.5299v4 [cs.LG]* .

KUNCAK, V. (2016) Parallel programming, Coursera MOOC. https://www.coursera.org/learn/parprog1, accessed July 2020.

KUO, L. AND MALLICK, B. (1998) Variable selection for regression models. , *The Indian Journal of Statistics: Series B (1960-2002)* **60** (1).

LAMBERT, B. (2018) *A Student's Guide to Bayesian Statistics*. Sage publications.

LAW, J. (2019) Scalable bayesian time series modelling for streaming data. Thesis submitted for the degree of Doctor of Philosophy-Newcastle University.

LAW, J. AND WILKINSON, D. J. (2019) Functional probabilistic programming for scalable Bayesian modelling. *arXiv:1908.02062v1 [stat.CO]* .

LAWLESS, C., WILKINSON, D., YOUNG, A., ADDINALL, S. AND LYDALL, D. (2010) Colonyzer: automated quantification of micro-organism growth characteristics on solid agar. *BMC Bioinformatics* **11** (287).

LAWVERE, W. (1962) The category of probabilistic mappings. https://ncatlab.org/nlab/files/lawvereprobability1962.pdf.

LEVY, D., PEREIRA-LEAL, B., CHOTHIA, C. AND TEICHMANN, A. (2006) 3D Complex: A Structural Classification of Protein Complexes. *PLoS Computational Biology* **2** (11), doi:[10.1371/journal.pcbi.0020155].

LI, S., TSO, K. AND LONG, L. (2017) Powered embarrassing parallel MCMC sampling in Bayesian inference, a weighted average intuition. *Computational Statistics and Data Analysis.* **115**, 11–20.

LINK, W. A. AND EATON, M. J. (2012) On thinning of chains in MCMC. *Methods in Ecology and Evolution* **3**, 112–115.

LIU, Q., LEE, J. AND JORDAN, M. (2016) A Kernelized Stein Discrepancy for Goodness-of-fit Tests. *arXiv:1602.03253v2 [stat.ML]* .

LODISH, H., BERK, A. AND ZIPURSKY, S. (2000) *Molecular Cell Biology*. New York: W. H. Freeman. 4th edition, available at: https://www.ncbi.nlm.nih.gov/books/NBK21475/.

LUNN, D., SPIEGELHALTER, D., THOMAS, A. AND BEST, N. (2009) The BUGS project: Evolution, critique and future directions. *Statistics in Medicine* **28**, 3049–3067.

MACEACHERN, S. N. AND BERLINER, L. M. (1994) Subsampling the Gibbs Sampler. *The American Statistician* **48** (3), 188–190.

Maciejowski, J. and deLange, T. (2017) Telomeres in cancer: tumour suppression and genome instability. *Nature Reviews Molecular Cell Biology volume* **18**, 175–186, https://doi.org/10.1038/nrm.2016.171.

Maclaurin, D. and Adams, P. R. (2014) Firefly Monte Carlo: Exact MCMC with subsets of data. *arXiv:1403.5693* 30th Conference on Uncertainty in Artificial Intelligence.

Maire, F., Friel, N. and Alquier, P. (2018) Informed Sub-Sampling MCMC: Approximate Bayesian Inference for Large Datasets. *arXiv:1706.08327v3 [stat.ME]* .

Mani, R., Onge, S., Hartman, J., Giaever, G. and Roth, F. (2008) Defining genetic interaction. *Proc Natl Acad Sci USA* 105: 34613466.

Martin, E. A. and Fu, A. Q. (2019) A Bayesian Approach to Directed Acyclic Graphs with a Candidate Graph. *arXiv:1909.10678v2[stat.ME]* .

Martin, G. M., Frazier, D. T. and Robert, C. P. (2020) Computing Bayes: Bayesian Computation from 1763 to 21st Century. *arXiv:2004.06425v1 [stat.CO]* .

van de Meent, J., Paige, B., Yang, H. and Wood, F. (2018) An Introduction to Probabilistic Programming. *arXiv:1809.10756v1 [stat.ML]* .

MITx (2013) Introduction to Biology The secret of Life. https://www.edx.org/course/introduction-to-biology-the-secret-of-life-3, accessed August 2020.

National Human Genome Research Institute (2020) Karyotype. https://www.genome.gov/genetics-glossary/Karyotype, accessed August 2020.

Neal, R. (2012) MCMC using Hamiltonian dynamics. *arXiv:1206.1901v1 [stat.CO]* .

Neiswanger, W., Wang, C. and Xing, E. (2014) Asymptotically Exact, Embarrassingly Parallel MCMC. *arXiv:1311.4780v2 [stat.ML]* .

Newman, K. (2017) Bayesian modelling of latent Gaussian models featuring variable selection. Thesis submitted for the degree of Doctor of Philosophy-Newcastle University.

NIMBLE (2020) Variable selection in NIMBLE using reversible jump MCMC. https://r-nimble.org/variable-selection-in-nimble-using-reversible-jump-mcmc, accessed October 2020.

Nishihara, R., Murray, I. and Adams, R. (2014) Parallel MCMC with Generalized Elliptical Slice Sampling. *Journal of Machine Learning Research* **15**, 2087–2112.

nLab (2020) Category theory. https://ncatlab.org/nlab/show/category+theory, accessed July 2020.

Oates, C. J., Girolami, M. and Chopin, N. (2016) Control Functionals for Monte Carlo Integration. *arXiv:1410.2392v5 [stat.ME]* .

Odersky, M. (2008) *Programming in Scala*. Artima, 1st edition, available online https://www.artima.com/pins1ed/, accessed July 2020.

Odersky, M. (2016) Functional Programming Principles in Scala, Coursera MOOC. https://www.coursera.org/learn/progfun1, accessed July 2020.

O'Hara, R. B. and Sillanpaa, M. J. (2009) A review of Bayesian variable selection methods what, how and which. *Bayesian analysis* **4** (1), 85–117.

Owen, A. B. (2017) Statistically efficient thinning of a Markov chain sampler. *arXiv:1510.07727v7 [stat.CO]* .

Paige, B., Sejdinovic, D. and Wood, F. (2016) Super-Sampling with a Reservoir. *UAI'16: Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence* pp. 567–576.

Papaspiliopoulos, O., Roberts, O. G. and Zanella, G. (2018) Scalable inference for crossed random effects models. *arXiv:1803.09460v1 [stat.CO]* .

Pfeffer, A. (2015) *Practical Probabilistic Programming*. Manning Publications.

Phenix, H., Morin, K., Batenchuk, C., Parker, J., Abedi, V., Yang, L., Tepliakova, L., Perkins, T. and Kærn, M. (2011) Quantitative Epistasis Analysis and Pathway Inference from Genetic Interaction Data. *PLoS Computational Biology* **7** (5), e1002048. doi:10.1371/journal.pcbi.1002048.

Piironen, J. and Vehtari, A. (2017) Sparsity information and regularization in the horseshoe and other shrinkage priors. *arXiv:1707.01694v1 [stat.ME]* .

Plummer, M. (2017) Just Another Gibbs Sampler. http://mcmc-jags.sourceforge.net/, accessed July 2020.

Quiroz, M., Villani, M., Kohn, R. and Tran, M. (2018) Speeding Up MCMC by Efficient Data Subsampling. *arXiv:1404.4178v6 [stat.ME]* .

Raftery, A. E. and Lewis, S. (1992) How Many Iterations in the Gibbs Sampler? pp. 763–773. Oxford University Press.

Rajaratnam, B. and Sparks, D. (2015) MCMC-Based Inference in the Era of Big Data: A Fundamental Analysis of the Convergence Complexity of High-Dimensional Chains. *arXiv:1508.00947v2 [math.ST]* .

Ramsey, N. and Pfeffer, A. (2002) Stochastic lambda calculus and monads of probability distributions. *ACM SIGPLAN Notices* **37**, 154–165, aCM.

Rendell, L. J., Johansen, A. M., Lee, A. and Whiteley, N. (2020) Global Consensus Monte Carlo. *arXiv:1807.09288v3 [stat.CO]* .

Riabiz, M., Chen, W., Cockayne, J., Swietach, P., Niederer, S. A., Mackey, L. and Oates, C. J. (2020) Optimal Thinning of MCMC Output. *arXiv:2005.03952v2 [stat.ME]* .

Roberts, G. O., Gelman, A. and Gilks, W. R. (1997) Weak convergence and optimal scaling of Random Walk Metropolis Algorithms. *The Annals of Applied Probability* **7** (1), 110–120.

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

Rue, H., Martino, S. and Chopin, N. (2009) Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations. *Journal of the Royal Statistical Society* **71** (2), 319–392.

Saccharomyces Genome Database (2020) Biological information for the budding yeast Saccharomyces cerevisiae. http://www.yeastgenome.org/, accessed November October 2020.

ScalaDocs (2020) Tour of scala. https://docs.scala-lang.org/tour/basics.html, accessed July 2020.

Scibior, A., Ghahramani, Z. and Gordon, A. D. (2015) Practical probabilistic programming with monads. *ACM SIGPLAN Notices* **50**, 165–176, aCM.

Scott, M. A., Simonoff, J. S. and Marx, B. D. (2013) *The SAGE Handbook of Multilevel Modeling*. Sage Publications.

Scott, S. (2017*a*) Comparing Consensus Monte Carlo Strategies for Distributed Bayesian Computation. *Brazillian Journal of Probability and Statistics* **31** (4), 668–685.

Scott, S., Blocker, A., Bonassi, F., Chipman, H., George, E. and McCulloch, R. (2016) Bayes and big data: The consensus Monte Carlo algorithm. *International Journal of Management Science and Engineering Management* **11**, 78–88.

Scott, S. O. (2017*b*) Comparing consensus Monte Carlo strategies for distributed bayesian computation **31** (4), 668–685.

Shammas, A. (2011) Telomeres, lifestyle, cancer, and aging. *Current Opinion in Clinical Nutrition and Metabolic Care* **14** (1), 28–34.

Spiegelhalter, D., Thomas, A., Best, N. and Lunn, D. (2003) Winbugs user manual. https://www.mrc-bsu.cam.ac.uk/wp-content/uploads/manual14.pdf, accessed October 2020.

Stan (2015) Stan. https://mc-stan.org/, accessed June 2020.

Stan (2019*a*) Divergent transitions after warmup. https://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup, accessed July 2020.

Stan (2019*b*) Effective sample size. https://mc-stan.org/docs/2_18/reference-manual/effective-sample-size-section.html, accessed July 2020.

Stan (2019*c*) Hamiltonian monte carlo. https://mc-stan.org/docs/2_21/reference-manual/hamiltonian-monte-carlo.html, accessed May 2020.

Stan (2019*d*) Maximum treedepth exceeded. https://mc-stan.org/misc/warnings.html#maximum-treedepth-exceeded, accessed July 2020.

Stan (2019*e*) Stan efficiency tuning. https://mc-stan.org/docs/2_18/stan-users-guide/optimization-chapter.html, accessed August 2020.

Sun, X. and Ni, L. (1990) Another view on parallel speedup/. *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* pp. 324–333.

Tan, L. S. L. and Nott, D. J. (2013) Variational Inference for Generalized Linear Mixed Models Using Partially Noncentered Parametrizations. *Statistical Science* **28** (2), 168–188.

Tan, Z., Roche, K., Zhou, X. and Mukherjee, S. (2018) Scalable Algorithms for Learning High-Dimensional Linear Mixed Models. *arXiv:1803.04431v1 [stat.ML]* .

Tong, A. H. and Boone, C. (2006) Synthetic genetic array analysis in saccharomyces cerevisiae. *Methods in Molecular Biology* **313**, 171–192, yeast Protocols, 2nd Edition.

Ungar, L., Yosef, N., Sela, Y., Sharan, R., E., E. R. and Kupiec, M. (2009) A genome-wide screen for essential yeast genes that affect telomere length maintenance. *Nucleic Acids Research* **37** (12), 3840–3849, doi: 10.1093/nar/gkp259.

VanDerwerken, N. D. and Schmidler, C. S. (2013) Parallel Markov Chain Monte Carlo. *arXiv:1312.7479v1 [stat.CO]* .

Vehtari, A., Gelman, A. and Gabry, J. (2016) Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *arXiv:1507.04544v5 [stat.CO]* .

Vono, M., Dobigeon, N. and Chainais, P. (2018) Split-and-augmented Gibbs sampler Application to large-scale inference problems. *arXiv:1804.05809v2 [stat.ME]* .

Wang, F., Wu, X., Essertel, G., Decker, J. and Rompf, T. (2019) Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *arXiv:1803.10228v3[cs.LG]* .

Wang, X. and Dunson, B. D. (2014) Parallelizing mcmc via weierstrass sampler. *arXiv:1312.4605v2 [stat.CO]* .

Wei, Z. and Conlon, M. E. (2017) Parallel Markov Chain Monte Carlo for Bayesian Hierarchical Models with Big Data, in Two Stages. *arXiv:1712.05907v1 [stat.ME]* .

Welch, P. and Heidelberger, P. (1983) Simulation run length control in presence of an initial transient. *Operations Research* **31** (6), 1109–1144.

Welling, M. and Teh, Y. (2011) Bayesian learning via stochastic gradient Langevin dynamics. *In Proceedings of the 28th International Conference on Machine Learning (ICML)* pp. 681–688.

Wikimedia Commons (2012) Eukaryote DNA and difference DNA-RNA. https://commons.wikimedia.org/wiki/File:Eukaryote_DNA-en.svg and https://commons.wikimedia.org/wiki/File:Difference_DNA_RNA-EN.svg, accessed August 2020.

Wilkinson, D. (2005) Parallel Bayesian Computation. Chapter 16 in E. J. Kontoghiorghes Handbook of Parallel Computing and Statistics.

Wilkinson, D. (2014) One-way ANOVA with fixed and random effects from a Bayesian perspective. https://darrenjw.wordpress.com/2014/12/22/one-way-anova-with-fixed-and-random-effects-from-a-bayesian-perspective/, accessed June 2020.

Wilkinson, D. J. and Yeung, S. K. H. (2004) A sparse matrix approach to Bayesian computation in large linear models, computational statistics and data analysis.

Wu, C. and Robert, C. (2017) Average of Recentered Parallel MCMC for Big Data. *arXiv:1706.04780v2 [stat.CO]* .

Wu, C., Stoehr, J. and Robert, C. P. (2015) Faster Hamiltonian Monte Carlo by Learning Leapfrog Scale. *arXiv:1810.04449v2 [stat.CO]* .