Bayesian estimation of decomposable Gaussian graphical models

.....

A thesis presented to

The University of New South Wales

in fulfillment of the thesis requirement
for the degree of

Doctor of Philosophy

by Helen Armstrong

21/12/05

Abstract

This thesis explains to statisticians what graphical models are and how to use them for statistical inference; in particular, how to use decomposable graphical models for efficient inference in covariance selection and multivariate regression problems. The first aim of the thesis is to show that decomposable graphical models are worth using within a Bayesian framework. The second aim is to make the techniques of graphical models fully accessible to statisticians.

To achieve these aims the thesis makes a number of statistical contributions. First, it proposes a new prior for decomposable graphs and a simulation methodology for estimating this prior. Second, it proposes a number of Markov chain Monte Carlo sampling schemes based on graphical techniques. The thesis also presents some new graphical results, and some existing results are reproved to make them more readily understood. Appendix 8.1 contains all the programs written to carry out the inference discussed in the thesis, together with both a summary of the theory on which they are based and a line by line description of how each routine works.

Acknowledgments

This thesis exists because of Robert Kohn's remarkably steadfast faith in my ability to complete it. On a more prosaic note, I would like to thank Robert for his invaluable academic advice and assistance, and for his commitment to seeing me through. Also my co-supervisors Chris Carter and Catherine Greenhill for many academic discussions and for performing the unenviable task of checking my graphical proofs. I would like to thank Ed Cripps and Farid Khoury for answering my many questions when I was learning to program. Finally, I would like to thank the many academics in Europe and America who answered my questions about their own research.

This thesis was completed with the financial assistance of an APA and scholarships from the School of Mathematics, and from an ARC grant funded scholarship from Robert Kohn.

Declaration

Contents

Chapter 1

Introduction

This thesis explains to statisticians what graphical models are and how to use them for statistical inference; in particular, how to use decomposable graphical models for efficient inference in covariance selection and multivariate regression problems. The first aim of the thesis is to show that decomposable graphical models are worth using within a Bayesian framework. The second aim is to make the techniques of graphical models fully accessible to statisticians.

To achieve these aims the thesis makes a number of statistical contributions. First, it proposes a new prior for decomposable graphs and a simulation methodology for estimating this prior. Second, it proposes a number of Markov chain sampling schemes based on graphical techniques. The thesis also presents some new graphical results, and some existing results are reproved to make them more readily understood. Appendix 8.1 contains all the programs written to carry out the inference discussed in the thesis, together with both a summary of the theory on which they are based and a line by line description of how each routine works.

The rest of Chapter 1 is organised as follows. Section 1.1 outlines the contributions of this thesis to the existing literature. Section 1.2 presents a statistical motivation for graphical models. Section 1.3 presents a summary review of the relevant literature.

1.1 Contributions

Estimating a covariance matrix efficiently is an important statistical problem with many applications, such as multivariate regression, cluster analysis, factor analysis, and discriminant analysis; see, for example, Mardia et al. (1979). Such applications are used in the fields of business, engineering, and the physical and social sciences. It is also of considerable interest to understand the graphical structure of the covariance matrix because it is directly interpretable in terms of the partial correlations of the underlying multivariate distribution. By the graph of the covariance matrix we mean the pattern of nonzero off-diagonal elements in the inverse of the covariance matrix, also called the concentration matrix (see Lauritzen 1996,Chapter 5). Estimating a covariance matrix efficiently and understanding its graphical structure is difficult because the number of unknown parameters in the covariance matrix increases quadratically with dimension, and because the estimate of the covariance matrix must be positive definite.

The literature review in Section 1.3 shows that whilst there is an extensive literature on Bayesian variable selection, model selection and model averaging, there is a significant gap in the literature in the area of multivariate covariance estimation using the efficiencies made possible by graphical methods.

There are three reasons for the slow adoption of graphical methods in statistics. First, the methods are difficult to understand and implement. Complicated software is required, which is not always readily available. Furthermore, texts on the subject assume a high degree of familiarity with fundamental graphical concepts that many statisticians lack. The people most familiar with the graph theory are predominantly computer scientists, who are more interested in structural learning and are unlikely to consider a statistical application for covariance estimation. Second, there is insufficient evidence demonstrating the advantages of applying graphical techniques to motivate statisticians to learn about them. Third, there is a lack of accessible literature explaining how graphical methods can be applied in Bayesian statistical inference in general, and for multivariate Gaussian regression models in particular.

This thesis considers Bayesian estimation of decomposable Gaussian covariance selection models, also known as decomposable graphical Gaussian models. Chapter 2 presents an introduction to graphical models and explains, from a statistical point

of view, why such models are interesting. It also explains how graphical methods can be used to facilitate statistical inference. In order to use the theory of Chapter 2, it is necessary to simulate an irreducible Markov chain in the state space of graphs. Chapter 3 explains how this can be done, and provides worked examples so that the theoretical results can be programmed easily. In the process, the thesis contributes some new graphical results that make transitions in the state space of decomposable graphs easier to understand and calculate. To make the theory easier to understand and implement for real inference, informal explanatory heuristics and statistical interpretations are given whenever relevant. New proofs of existing fundamental results together with illustrative examples of the application of the proof are provided whenever such a proof makes the result easier to understand.

Another major contribution of this thesis is to implement graphical techniques in a statistical framework: specifically, to provide empirical evidence that they are an efficient way to perform covariance estimation, and show how they can be applied in general regression models for variable and covariance selection. This is achieved as follows.

First, a prior for the covariance matrix is proposed such that the probability of each graph size is specified by the user, where the size of a graph is defined as the number of its edges. Most previous approaches, e.g. Giudici & Green (1999), assume that all graphs are equally probable. Section 7.2 reports the results of a simulation study that shows that the prior that assigns equal probability over graph sizes outperforms the prior that assigns equal probability over all graphs, both in identifying the correct decomposable model and in estimating the covariance matrix more efficiently. This advantage is greatest when the number of observations is small relative to the dimension of the covariance matrix. The simulations in Section 4.14 suggest that there is relatively little loss in statistical efficiency in using mixtures of decomposable models compared to the estimator of Wong et al. (2003), even when the graph of the covariance matrix is not decomposable.

The new prior uses the counts $A_{p,k}$ of decomposable graphs of size k with p vertices. The next contribution is to propose a Markov chain Monte Carlo method for estimating these counts and to show that the counts obtained by the simulation method agree with analytic results when such results are known. Chapter 7 evaluates and assesses the new prior for the decomposable space. It presents a simulation

methodology for estimating the prior, and gives the results of the simulation. The numbers produced have not been calculated before because it is beyond current computational limits to do so analytically for $p \geq 9$. The numbers for $p \leq 8$ are calculated exactly, and are presented because they are not readily available in the literature at present.

Chapter 4 explains Bayesian covariance selection models, and gives some results for the graph dependent version of the Wishart and inverse Wishart distributions, as these are used in the Markov chain Monte Carlo simulations. Section 4.8 shows how to use the marginal likelihood results in Giudici (1996) to derive a reduced conditional MCMC sampler for decomposable graphical models, where the covariance matrix is integrated out of all conditional distributions and is not generated in the Markov chain Monte Carlo . This result is based on Wong (2002). This approach does not require reversible jump Metropolis-Hasting methods and has the local computation properties of the Giudici & Green (1999) approach, so the computational complexity for one iteration of the approach in this thesis is similar to that of Giudici & Green (1999). However, it is reasonable to expect that the sampler in this thesis has a faster convergence rate to that of Giudici & Green (1999) and Brooks et al. (2003), but we have not compared them emprically.

Sections 4.3 and 4.11 show how the exact posterior results of Dawid & Lauritzen (1993) and Roverato & Whittaker (1998) for graph dependent versions of the Wishart and inverse Wishart distributed covariance matrix can be used together with the results in Roverato (2000) to sample graph dependent covariance matrices directly from their exact posterior distribution. In addition to the presentation of the theory, detailed worked examples are used to illustrate and explain the difficult graphical concepts involved.

Section 4.14 compares the performance of the reduced conditional sampler (which assumes a decomposable prior) to the more general, but less efficient, prior of Wong et al. (2003). This section provides empirical evidence that the decomposable reduced conditional sampler has a faster convergence rate than the Wong et al. (2003) approach. The results in this section suggest that at present there is no 'best' method for estimating Gaussian covariance selection models. While the method of Wong et al. (2003) works in principle for all graphs, the convergence of their MCMC simulation can be slow if the true graph has full subgraphs of size 5 or larger because

Wong et al. (2003) generate the elements of the concentration matrix one at a time. On the other hand, the decomposable reduced conditional sampler is extremely efficient because the concentration matrix is integrated out. It is therefore an attractive alternative to the Wong et al. (2003) model for high dimensional graphs that are likely to have substantial full subgraphs. Another advantage of the decomposable prior is that there is a separate normalizing constant for each decomposable graph, whereas Wong et al. (2003) have a normalizing constant for each graph size. A comparison between the MCMC methods described in this thesis and a stochastic search approach for finding the graph with maximum posterior probability is described in Jones et al. (2005).

Chapter 5 applies the decomposable graphical methdology for covariance selection in multivariate regression, and compares it to the results of Cripps et al. (2005) who use the nondecomposable methdology of Wong et al. (2003). In this chapter variable selection is also carried out on the regression coefficients. Chapter 5 also shows how the graphical methodology allows for a far richer interpretation of the data than is possible using conventional methods. It also contains analysis of higher dimensional datasets. The number of iterations required to analyse these datasets using the Wong et al. (2003) or Cripps et al. (2005) methodologies may become prohibitive when the inverse covariance is not sparse, because of the high autocorrelations in the iterates.

Chapter 6 proposes a new sampling scheme for covariance selection and variable selection in a multivariate regression model that integrates out both the regression coefficients and the covariance matrix. Such a sampling scheme is not possible using the methods of Cripps et al. (2005). Section 6.4 gives empirical evidence that the sampler in Chapter 6 is more efficient than both the sampler of Cripps et al. (2005) and the decomposable sampler of Chapter 5 on a number of real datasets.

The final contribution of this thesis is a complete graphical analysis package, in MATLAB and FORTRAN, that is sufficient for performing all the analysis presented in this thesis. Each subsection of Appendix 8.1 corresponds to a single routine. It gives an explanation of the routine and the theory on which it is based, followed by a line by line description of the algorithms involved in each step. These programs and explanations are given so that statisticians can apply the techniques of this thesis easily.

1.2 Motivation

Suppose we have independent observations

$$y_t \sim N(\mu, \Sigma), \quad t = 1, \ldots, n, \tag{1.1}$$

where y_t is $p \times 1$ and Σ is the covariance matrix. Let $y = (y_1, \ldots, y_n)$ be the data. Suppose our aim is to estimate a model for y. Estimating a covariance matrix efficiently is difficult because the number of unknown parameters in the covariance matrix increases quadratically with dimension, and because the estimate of the covariance matrix must be positive definite.

A more parsimonious model results if there are zeros in the covariance matrix Σ or the concentration matrix $\Omega = \Sigma^{-1}$, and for Gaussian data such zeros have the following interpretation. If $\Sigma_{ij} = 0$, then y_{it} and y_{jt} are independent for all t. We also have that if $\Omega_{ij} = 0$ then y_{it} and y_{jt} are independent conditional on y_{kt} for all $k \neq i, j$ (See Wermuth (1976)). We note that allowing for zeros in the concentration or covariance matrix can improve the statistical efficiency of the estimated covariance matrix, as well as improve inference for the multivariate linear regression model, because the predictive distribution is estimated more efficiently.

However, selecting which elements of Ω to set to zero is difficult even for moderate dimensions because a $p \times p$ concentration matrix has $k = p(p-1)/2$ distinct off-diagonal entries and there are 2^k possible possible configurations of concentration matrices with zero elements associated with it. A better approach is to consider subset spaces which become increasingly sparse as p increases. It is well known that the space of decomposable matrices is such a space, and for completeness this is proved in Appendix 8.3.

1.3 Literature review

There is a large literature of methods that use shrinkage or Bayesian models to improve on the maximum likelihood estimator of the covariance matrix. See, for example, Dempster (1969), Dempster (1972), Efron & Morris (1976), Yang & Berger (1994), Chiu et al. (1996), Giudici & Green (1999), Barnard et al. (2000), Wong et al. (2003) and Liechty et al. (2004). The simulation studies in Yang & Berger (1994) and Wong et al. (2003) show that considerable gains in efficiency are possible.

Dempster (1972) advocates a covariance selection approach to estimate a covariance matrix more efficiently, by which he means setting to zero some of the off-diagonal elements of the concentration matrix. His idea is that a more parsimonious model will give greater efficiency. As mentioned in the previous section, the selection of which elements to set to zero is difficult even for moderate dimensions because a $p \times p$ concentration matrix has $p(p-1)/2$ distinct off-diagonal entries and there are $2^{p(p-1)/2}$ possible graphs associated with it. Drton & Perlman (2004) give a model selection approach based on simultaneous confidence intervals to determine which partial correlations are zero. The simultaneous confidence intervals are based on large sample theory and become large when p is moderate to large. Drton & Perlman (2004) do not attempt to estimate the covariance matrix based on their selected graph.

A number of articles take a Bayesian approach to covariance selection. For the case of decomposable graphs, Dawid & Lauritzen (1993) introduce a conjugate prior for the covariance matrix called the hyper inverse Wishart distribution. Giudici (1996) uses a prior for the covariance matrix that is a mixture of fixed parameter hyper inverse Wishart priors over decomposable graphs and calculates the marginal likelihood for each decomposable graph, up to an overall normalizing constant. The marginal likelihood is used to calculate the posterior probability of each graph. This gives an exact solution for small examples, but for p greater than approximately 8 the number of graphs is prohibitively large.

Roverato (2000) shows that the hyper inverse Wishart prior for the covariance matrix is equivalent to a constrained Wishart prior for the concentration matrix. It is straightforward to define a constrained Wishart prior for general graphs, however, such distributions have normalizing constants that are not available analytically unless the graph is decomposable. Roverato (2002), Atay-Kayis & Massam (2005) and Dellaportas et al. (2004) propose efficient simulation and importance sampling methods for estimating the normalizing constants for the nondecomposable graphs. The normalizing constants are used to examine a small number of graphs and select those that that have the highest marginal likelihood or posterior probability, rather than to estimate the covariance matrix by averaging over graphs. However, such an approach seems unsuitable as the basis of a Markov chain Monte Carlo sampling scheme when p is moderate to large because there are $2^{p(p-1)/2}$ possible graphs with

only a small fraction of them being decomposable.

Giudici & Green (1999) give a MCMC approach that can deal with large values of p. Their method applies to a hierarchical model with a hyper inverse Wishart prior for the covariance matrix conditional on a decomposable graph. They use reversible jump Metropolis-Hastings methods to generate the covariance matrix and other parameters. Their method has a local computation property that only requires Cholesky decompositions of the submatrix of the covariance matix corresponding to a clique of the graph. Brooks et al. (2003) modify the reversible jump MCMC proposal of Giudici & Green (1999) and give empirical results to show this improves the convergence rate.

Wong et al. (2003) also use MCMC methods to select which off-diagonal elements to set to zero. They use reversible jump Metropolis-Hastings methods to generate the inverse covariance matrix and other parameters. The main difference between Giudici & Green (1999) and Wong et al. (2003) is that Wong et al. (2003) do not constrain the possible graphs to be decomposable. Wong et al. (2003) use a prior with normalizing constants based on graph size to avoid having to calculate normalizing constants for each nondecomposable graph. They also need to run a separate MCMC to estimate the normalizing constants for each graph size.

For longitudinal data, Smith & Kohn (2002) factor the concentration matrix using a Cholesky decomposition and carry out variable selection on the strict lower triangle of the Cholesky factor to obtain parsimony. Their approach is attractive when there is some natural ordering of the observation vector, but there are two potential drawbacks to the Cholesky approach when such a natural ordering does not exist. First, different orderings of the variables can yield different estimates of the covariance matrix. Second, under some orderings the Cholesky factor may be quite full even if the concentration matrix is sparse.

Chapter 2

Introduction to Gaussian graphical models

2.1 Introduction

This chapter introduces graphical models and explains why such models are interesting. To do so, some definitions from graph theory are given in Section 2.2. Section 2.3 shows how the concepts of graph theory can be used to represent the likelihood as a product of low dimensional terms with some nice properties. In the case of Gaussian distributions, we show how to associate a graph with the inverse covariance matrix to derive the representation. Section 2.5 discusses the theory necessary to find a decomposable version of any positive definite matrix, and hence how to define decomposable versions of Gaussian distributions. Conversely, Section 2.6 shows how the same graph-theoretic concepts can be used to define complex multivariate distributions with desirable independence properties, which are constructed from simple lower-dimensional distributions. Together these results show how graphs can be used as an inference tool for simplifying and organising probability calculations. Thus the theory of graphical models can be used to facilitate statistical inference in general, and covariance selection in particular.

More detail on the material in this chapter can be found in (Lauritzen, 1996, Chapters 2 and 3).

2.2 Graph theory

This section gives some definitions and results from graph theory that are fundamental to Gaussian graphical models. It is intended for reference only, as detailed explanations using illustrative examples are given in Chapter 3.

Let V be a finite set of *vertices*. Let $E = \{(u, v) : u \in V, v \in V, u \neq v\}$ be a set of *edges* so that $E \subset V \times V$. Define a graph g as an ordered pair $g = (V, E)$ of vertices and edges, where V is assumed to be finite. Write $v \in g$ whenever $v \in V$, and similarly $e \in g$ to denote any $e \in E$. For any graph C, write $g + C$ to denote the graph obtained by adding all edges and vertices in C to g. Since an edge (u, v) is equivalently the graph $e = (V = \{u, v\}, E = \{(u, v), (v, u)\})$, any subset of edges $E' \subset E$ can be used to define a *subgraph* $g' = (V, E')$ of g, and we can write $g + e$ for the graph obtained by adding e to g. For any subgraph or subset of edges $C \subset g$, or any set of vertices C, write $g - C$ to denote the subgraph obtained by deleting from g all vertices in C and all edges in g with at least one vertex in C. For any $A \subset V$ the *induced subgraph* is defined as the graph $g_A = (A, E_A)$ formed by keeping only those edges in g with both endpoints in A. Write $u \sim v$ if either (u, v) or $(v, u) \in E$, and say u and v are *adjacent*. It is assumed throughout this thesis that g is a *simple undirected graph*. That is, without loops (so $u \neq v$ for all u, v such that $(u, v) \in E$) or multiple edges (so every element in E is distinct), and $(v, u) \in E$ whenever $(u, v) \in E$. Define the set of *neighbours* (denoted by $nbrs(v)$) of a vertex v in g as the set of all vertices that are adjacent to v in g. Define $[v] = \{v\} \cup nbrs(v)$ as the *closure* of v in g obtained as the union of v with its neighbours. A subset $K \subseteq V$ is called *complete* if every pair of vertices are adjacent. Define a *clique* of g as a maximal complete subgraph of g. By maximal we mean that a clique is not contained in a larger complete subgraph.

For any sequence $B_1, \ldots B_k$ of subsets of V, define the *histories* of the sequence as the sets $H_j = \cup_{i=1}^{j} B_i$ for $j = 1, \ldots, k$. Similarly define the *separators* of the sequence as $S_j = B_j \cap H_{j-1}$ for $j = 2, \ldots, k$ and the *residuals* of the sequence as $R_j = B_j \backslash H_{j-1}$ for $j = 2, \ldots, k$.

We say that the sequence $\{B_1, \ldots B_k\}$ satisfies the *running intersection property* if for all $i > 1$ there is a $j < i$ such that $S_i \subseteq B_j$. The sequence is called *perfect* if it satisfies the running intersection property, and all the sets $S_i, i = 2, \ldots, k$ are

complete. Section 2.4 shows that when V is the index set for a random vector X_V, the existence of a perfect sequence of sets makes it easy to find a representation of the likelihood in terms of low dimensional factors.

A numbering $v_1, \ldots, v_p$ of the vertices V of $g = (V, E)$ is called a *perfect numbering* if $B_j = [v] \cap \{v_1, \ldots, v_j\}$ is a perfect sequence. (In this case $B_1, \ldots, B_k$ are all complete, by definition.) We will see in Section 3.5 that this numbering can be found recursively in an algorithmic process that aborts if there exists no perfect sequence of complete sets. On the other hand, if the numbering does exist, it can be used to find the perfect sequence. Hence perfect numberings are critical in finding representations of the likelihood in terms of low dimensional factors that facilitate posterior statistical inference.

Note that we follow the conventions of Lauritzen (1996), where sequences of individual set elements are perfect *numberings*, and the associated sequences of sets are perfect *sequences*.

Let g be a graph having p vertices labelled $v_1, \ldots, v_p$. Then the *adjacency matrix* G of g is the $p \times p$ matrix whose ij entry $G_{ij} = 1$ if $e = (v_i, v_j) \in E$, and $G_{ij} = 0$ otherwise. It is assumed throughout that all graphs are simple, so the diagonal elements G_{ii} are all zero. A *path* of *length* k in g from a to b is an alternating sequence of its $k-1$ vertices and k edges of the form $v_0, e_1, v_1, e_2, \ldots, e_k, v_k$, where vertices v_{i-1} and v_i are endpoints of edge e_i for each i, and all the vertices are distinct. Since an edge is uniquely characterised by its endpoints, we may denote paths by the sequence of vertices only. The *path matrix* of a graph g is the adjacency matrix of the transitive closure of g. That is, the path matrix G' of g satisfies $G'_{ij} = 1$ if and only if there is a path of any length from v_i to v_j in g, and $G'_{ij} = 0$ otherwise.

Define the *distance* between vertices u and v in a graph g as the length of the shortest path between them, denoted $d_g(u, v)$. Any pair of vertices joined by a path are *connected*. If all vertices are connected then g is said to be connected. Otherwise g will consist of connected components which are maximal connected subgraphs of g.

Refer to the graph in Figure 3.3 as $g5$. In $g5$, vertices 1 and 6 are adjacent, as are vertices 6 and 4. These characterise the edges $e_1 = (1, 6)$ and $e_2 = (6, 4)$ respectively. Vertices 1 and 4 are not adjacent, but there is a path between them. There is a path between every pair of vertices in $g5$. Hence $g5$ is connected and consists of a single

connected component. However, the graph we shall refer to as $g26$, depicted in Figure 2.9, is not connected, and consists of 3 connected components.

For any two vertices a, b, we say that a subset S is an *(a,b)-separator*, or that S *separates* a and b, if every path between a and b includes at least one vertex $s \in S$. Write $a \overset{g}{\perp} b|S$ for this, and omit g when the context is clear. Similarly, write $A \overset{g}{\perp} B|S$ and say that S separates the sets A and B if for every vertex pair $a \in A$ and $b \in B$, each path connecting a and b includes at least one element of S.

Any path that begins and ends at the same vertex (but in which every other vertex is distinct) is called a *cycle*. If the cycle involves n distinct edges then it is called an n-cycle. A *tree* is a connected graph that has no cycles. A *forest* is a graph (connected or not connected) that has no cycles. Define a *chord* of a cycle as a pair of vertices that are not consecutive on the cycle, but which are adjacent in g. We say that a graph is *chordal* if every n-cycle, $n \geq 4$, has at least one chord.

Define a *decomposition* of $g = (V, E)$ as a pair (A, B) of subsets of V such that $V = A \cup B$, $A \cap B$ is complete, and $A \cap B$ separates A from B. We say (A, B) *decomposes* g into the induced component subgraphs g_A and g_B. Define a *proper decomposition* as one in which both A and B are proper subsets of V.

We say that a graph g is *decomposable* if it is complete, or if there exists a proper decomposition (A, B) into decomposable subgraphs g_A and g_B. (pp. 1310-11, Dawid & Lauritzen (1993)).

An important advantage of decomposable over nondecomposable graphs is that their vertices can be arranged in a perfect numbering, or, equivalently, their cliques can be arranged in a perfect sequence. This is summarised in the proposition given below. This is an advantage because perfect numberings and sequences are easy to determine using algorithmic procedures, as illustrated in Section 3.5.3 and will be shown to be very useful for calculations in subsequent sections.

Proposition 2.2.1 *(Lauritzen, 1996,p.18). The following conditions are equivalent for an undirected graph g:*

1. *the vertices of g admit a perfect numbering;*
2. *the cliques of g can be numbered to give a perfect sequence;*
3. *the graph g is decomposable.*

A second advantage of decomposable models is that decomposability is equivalent to chordality (see Leimer (1989)). This is an advantage because chordality is easy to determine using algorithmic procedures, as illustrated in Chapter 3.

2.3 Statistical graphical models

Let X_V be a random vector whose elements are indexed by the finite set V, with $p = |V|$ the number of elements of V. Let $p_V(x_V)$ be the joint density of X_V and P_V the associated probability distribution. If X_V is Gaussian we write $X_V \sim N_{|V|}(\mu, \Sigma)$ and let $\Omega = \Sigma^{-1}$ throughout.

For any $A \subset V$, let X_A be the subvector $(X_a : a \in A)$ of X_V and let $p_A(x_A)$ be its density. If $R, S \subseteq V$ and $A = R \cup S$, then $X_A = X_{R \cup S}$ is referred to as the *union* of random vectors X_R and X_S, and we write $X_A = X_R \cup X_S$. We say it is a *disjoint union* if $A = R \cup S$ is a disjoint union.

We use the following notation of Dawid (1979), which has become standard. Let A, B, C be any subsets of V. For random subvectors X_A, X_B, X_C of X_V we write $X_A \perp\!\!\!\perp X_B | X_C \, [P_V]$ if X_A and X_B are independent conditional on X_C. We write $\perp\!\!\!\perp \{X_A, X_B, X_C\} \, [P_V]$ to indicate the mutual independence of all three, and $X_A \perp\!\!\!\perp X_B \, [P_V]$ for the marginal independence. Indication of the distribution $[P_V]$ is omitted when it is clear from the context.

We call a random vector X_A *complete* if there are no independencies or conditional independencies between any of the random variables comprising X_A, and say that it is *maximally complete* if it is not a proper subvector of any complete random vector.

For any $g = (V, E)$, a *statistical graphical model* $\mathfrak{g}$ is a family of distributions for the collection of random variables $X_V = (X_v)_{v \in V}$ indexed by the elements of V and taking values in probability spaces $\mathfrak{X}_V = (\mathfrak{X}_v)_{v \in V}$, where each member of the family satisfies $X_A \perp\!\!\!\perp X_B | X_D$ for every triple $A, B, D \subseteq V$ such that $A \overset{g}{\perp} B | D$. Thus g is a representation of the structure of independencies satisfied by every member of $\mathfrak{g}$. A *Gaussian graphical model* is a graphical statistical model in which every member is Gaussian. It is clear that the graphical condition of separation in an undirected graph cannot represent any distribution in which there are induced dependencies, so graphical statistical models cannot be used to represent all distributions. For example, if $P_V \in \mathfrak{g}$ for the graph depicted in Figure 3.7, then $X_b \perp\!\!\!\perp X_j | X_c$, and

$X_b \perp\!\!\!\perp X_j | X_c, X_U$ for any subset of variables U.

A probability measure P on $\mathfrak{X}_V$ is said to be *globally Markov*, or to obey the *global Markov property*, relative to g, if $X_A \perp\!\!\!\perp X_B | X_D$ for any triple of subsets A, B, D such that $A \overset{g}{\perp} B | D$. A density p_V is similarly defined as *globally Markov*, or to obey the *global Markov property*, relative to g, if its associated measure is globally Markov. If D is null, then $X_A \perp\!\!\!\perp X_B | X_D$ means that X_A and X_B are independent. This is equivalent to $A \overset{g}{\perp} B | D$ which means that A and B are not connected, so disconnectedness defines independence in a statistical graphical model.

If X_V is Gaussian, define the graph $g(\Omega)$ as having vertices V, with the edge set $E = (u, v) \subset V \times V$ such that $u \neq v$ and $\Omega_{u,v} \neq 0$. It is well known (e.g. Proposition 5.2, p. 129, Lauritzen (1996)) that $\Omega_{u,v} = 0$ if and only if $X_u \perp\!\!\!\perp X_v | X_{V \setminus \{u,v\}}$. The next proposition shows that for a Gaussian model, P_V is globally Markov with respect to $g(\Omega)$.

Proposition 2.3.1 *(Proposition 2, Speed & Kiiveri (1986)). Let $g = (V, E)$ be an undirected graph with vertex set V indexing the Gaussian random variables $X_V \sim N_{|V|}(\mu, \Sigma)$. Let $\Omega = (\Sigma)^{-1}$. Then the following are equivalent.*

(i) $\Omega_{\alpha\beta} = 0$ if $(\alpha, \beta) \notin E$ and $\alpha \neq \beta$;

(ii) for every $v \in V$, $X_v \perp\!\!\!\perp X_{V \setminus v} | X_{nbrs(v)}$; and

(iii) $X_A \perp\!\!\!\perp X_B | X_D$ for any triple of subsets A, B, D such that $A \overset{g}{\perp} B | D$.

2.4 Decomposable statistical graphical models

This section defines a decomposable statistical graphical model and shows that for such a model, the density of X_V is a product of lower dimensional clique dependent densities. This allows for efficient inference. Furthermore, the cliques are easy to determine (see Section 3.5.3), and so decomposable models simplify statistical computations.

Deriving a model for p_V based on a set of observations (data), is facilitated when the likelihood can be represented as a product $p_V(x_V) = \prod_{B \in \mathcal{B}} p(x_B | x_{S_B}), B, S \subset V$ of low dimensional factors $p(x_B | x_{S_B})$, such that $\sum_{B \in \mathcal{B}} |S_B|$ is minimal amongst

all factorisations of $p_V(x_V)$, and $V = \cup_{B\in\mathcal{B}} B$ is a disjoint union. We call such a factorisation a *minimal factorisation*, if it exists.

The next lemma summarises an important advantage of decomposable models: that finding a low dimensional factorisation is equivalent to finding a perfect sequence of cliques (which is computationally simple, as explained in Section 3.5.3).

Lemma 2.4.1 *Let $g = (V, E)$ be a decomposable graph with perfect sequence of cliques $\{C_1, \ldots, C_k\}$, separators $S_i = C_i \cap H_i$, and residuals $R_i = C_i \backslash H_{i-1}$. Then any density p that is globally Markov with respect to g factorises according to g as*

$$p(x_V) = \prod_{i=1}^{k} p(x_{R_i}|x_{S_i})p(x_{C_1}). \tag{2.1}$$

Proof. By Lemma 2.10.1 (in the appendix to this chapter), $X_{R_{i+1}} \perp\!\!\!\perp X_{H_i \backslash S_{i+1}} | X_{S_{i+1}} [P_V]$. Thus,

$$\begin{aligned} p_V(x_{H_{i+1}}) &= p_V(x_{H_{i+1}\backslash H_i}|x_{H_i})p_V(x_{H_i}) \\ &= p_V(x_{R_{i+1}}|x_{H_i})p_V(x_{H_i}) \\ &= p_V(x_{R_{i+1}}|x_{H_i\backslash S_{i+1}}, x_{S_{i+1}})p_V(x_{H_i}) \\ &= p_V(x_{R_{i+1}}|x_{S_{i+1}})p_V(x_{H_i}) \text{ since } X_{R_{i+1}} \perp\!\!\!\perp X_{H_i\backslash S_{i+1}} | X_{S_{i+1}} [P_V]. \end{aligned} \tag{2.2}$$

As $p(x_{H_1}) = p(x_{C_1})$, the proof is obtained by induction and that Lemma 2.10.1 guarantees all independences hold in P_V, not just $P_{H_{i+1}}$. ■

We can use Lemma 2.4.1 to derive the well known result that expresses p_V in terms of its clique and separator marginals. We use this lemma throughout for Bayesian inference.

Lemma 2.4.2 *(Equation (5.44), p. 144, Lauritzen (1996)). Let p_V be a joint density which is globally Markov with respect to decomposable $g = (V, E)$. Let $\mathcal{C}$ and $\mathcal{S}$ be the cliques and separators of g, with corresponding marginal densities $\{p_C : C \in \mathcal{C}\}$ and $\{p_S : S \in \mathcal{S}\}$, respectively.*

Then p_V factorises

$$p_V(x_V) = \frac{\prod_{C\in\mathcal{C}} p_C(x_C)}{\prod_{S\in\mathcal{S}} p_S(x_S)}. \tag{2.3}$$

Proof. By definition, $C_i = R_i \cup S_i$, and so $p(x_{R_i}|x_{S_i}) = p(x_{R_i \cup S_i})/p(x_{S_i}) = p(x_{C_i})/p(x_{S_i})$. Substituting $p(x_{C_i})/p(x_{S_i})$ for $p(x_{R_i}|x_{S_i})$ in (2.1) gives the result. ■

We remark that the equivalence of these two factorisations is also given by the uniqueness of the distribution in Lemma 2.6.4, together with Proposition 2.6.1 and Proposition 2.6.2.

2.5 Calculating a g-constrained covariance matrix

This section discusses the theory necessary to find a decomposable version of any positive definite matrix. This allows us to define decomposable subfamilies of Gaussian distributions.

Given any positive definite matrix Σ and decomposable graph g with adjacency matrix G, we require a positive definite matrix $\Sigma|g$ such that $(\Sigma|g)_{ij} = \Sigma_{ij}$ if $G_{ij} = 1$ or $i = j$, and $(\Sigma|g)^{-1}_{ij} = 0$ if $G_{ij} = 0$. Note that the entries $(\Sigma|g)_{ij}$ for which $G_{ij} = 0$ are left undefined by this requirement. If such a $\Sigma|g$ is uniquely defined by the requirement, then we say it is the *g-constrained version* of Σ.

Theorem 1 of Speed & Kiiveri (1986) asserts the existence and uniqueness of $\Sigma|g$. (Existence and uniqueness also follows from the result of Grone et al. (1984) on the existence and uniqueness of the positive completion.) We introduce the following notation. Define the *complementary graph* of $g = (V, E)$ as $\tilde{g} = (V, \tilde{E})$ where $\tilde{E} = (V \times V) \backslash E$ has the property that $(u, v) \in \tilde{E}$ if and only if $u \neq v$ and $(u, v) \notin E$. Denote by $\tilde{\mathcal{C}}$ a set of cliques of $\tilde{g}$.

Theorem 2.5.1 *(Speed & Kiiveri, 1986, Theorem 1) Given positive definite matrices L and M defined on the vertices V of a graph $g = (V, E)$, there exists a unique positive definite matrix K such that*

1. *$K_{uv} = L_{uv}$ if $(u, v) \in E$ or $u = v$.*
2. *$(K)^{-1}_{uv} = M_{uv}$ if $(u, v) \notin E$ and $u \neq v$.*

Equivalently,

1. *$K_C = L_C$ if $C \in \mathcal{C}$.*

2. For any $\tilde{C} \in \tilde{\mathcal{C}}$, $(K)^{-1}_{\tilde{C},\tilde{C}}$ and $M_{\tilde{C},\tilde{C}}$ agree except on the diagonals.

Setting $K = \Sigma|g$, $L = \Sigma$ and $M = G$ in Theorem 2.5.1 gives a natural parameterisation of g so that g can be associated with a *full* covariance Σ. So, in the zero mean Gaussian case, in which the covariance defines the distribution, g can be associated with a full distribution, not just its conditional independencies. That is, every covariance has a uniquely defined g-constrained version with the same entries on the edge set of g, and with an inverse that has the same zero pattern as G except on the diagonals.

Any decomposable graph g can be used as the basis of an efficient algorithm for computing $\Sigma|g$ as follows. Let $M^+(g)$ be the set of $|V| \times |V|$ symmetric positive matrices Ω satisfying $\Omega_{uv} = 0$ for all $u \nsim v$ in g. For any submatrix Ω_{AB} of Ω on the vertices in $A, B \subset V$ denote by $[\Omega_{AB}]^V$ the $|V| \times |V|$ matrix whose remaining entries are zero. That is, $[\Omega_{AB}]^V_{AB} = \Omega_{AB}$ and $[\Omega_{AB}]^V_{uv} = 0$ for all $u \notin A$, $v \notin B$.

Lemma 2.5.2 *(Lauritzen, 1996,Lemma 5.5, p. 136) Let $\Omega \in M^+(g)$, and let (A, B, C) be a disjoint partitioning of g with C separating A from B. Then*

$$\Omega = [\Omega_{A\cup C}]^V + [\Omega_{B\cup C}]^V - [\Omega_C]^V \tag{2.4}$$

and for any symmetric $|V| \times |V|$ matrix L we have

$$tr(\Omega L) = tr(\Omega_{A\cup C} L_{A\cup C}) + tr(\Omega_{B\cup C} L_{B\cup C}) - tr(\Omega_C L_C). \tag{2.5}$$

If Ω is invertible and $\Sigma = \Omega^{-1}$ then

$$\Omega = \left[(\Sigma_{A\cup C})^{-1}\right]^V + \left[(\Sigma_{B\cup C})^{-1}\right]^V - \left[(\Sigma_C)^{-1}\right]^V \tag{2.6}$$

and the determinant satisfies

$$det(\Omega) = det((\Sigma_C))/det(\Sigma_{A\cup C})det(\Sigma_{B\cup C}). \tag{2.7}$$

Repeated use of Lemma 2.5.2 gives $\Omega = (\Sigma|g)^{-1}$ as follows:

$$\Omega = \sum_{C\in\mathcal{C}} [\Omega_{CC}]^V - \sum_{S\in\mathcal{S}} [\Omega_{SS}]^V \tag{2.8}$$

$$= \sum_{C\in\mathcal{C}} \left[(\Sigma_{CC})^{-1}\right]^V - \sum_{S\in\mathcal{S}} \left[(\Sigma_{SS})^{-1}\right]^V, \tag{2.9}$$

where the sum over the separators includes each possibly nondistinct $S_i = C_i \cap H_{i-1}$.

The maximum likelihood estimate of Ω for decomposable covariance selection models is given by an analogous repeated use of Proposition 5.6, Equation 5.29, p. 138, Lauritzen (1996):

$$\Omega = \sum_{C \in \mathcal{C}} \left[((S_y)_{CC})^{-1} \right]^V - \sum_{S \in \mathcal{S}} \left[((S_y)_{SS})^{-1} \right]^V . \tag{2.10}$$

where the sum over the separators again includes each possibly nondistinct $S_i = C_i \cap H_{i-1}$, and $S_y = \sum_{t=1}^{n} (y_t - \overline{y})(y_t - \overline{y})'$.

If g is not decomposable, then $\Sigma|g$ can not be found efficiently (see p. 134, Lauritzen (1996)).

2.6 Constructing multivariate distributions from decomposable graphs

This section shows that for any decomposable graph, it is possible to define complicated statistical models from simpler clique dependent distributions. Furthermore, these distributions are completely characterised by the clique dependent distributions.

Because zero mean Gaussian measures are completely specified by the covariance matrix, specifying clique marginal submatrices of the covariance matrix is equivalent to specifying clique marginal densities p_{C_i}. Thus Theorem 2.5.1 in Section 2.5 characterises the Gaussian graphical model $\mathfrak{g}$ of Gaussian statistical measures in terms of the consistent clique marginal densities p_{C_i} corresponding to the submatrices Σ_{C_i,C_i}. This is a surprising result, as it leaves unspecified the entries $\Sigma_{u,v}$ for which $(u, v) \notin E$. This property of the hyper inverse Wishart is explained in more detail in Section 4.2.

We begin with some necessary definitions. A probability measure P on $\mathcal{X}$ is said to be *pairwise Markov*, or to obey the *pairwise Markov property*, relative to any graph g, if for any pair u, v of nonadjacent vertices $X_u \perp\!\!\!\perp X_v | V \backslash \{u, v\}$.

A probability measure P on $\mathcal{X}$ is said to be *locally Markov*, or to obey the *local Markov property*, relative to g, if for every $v \in V$, $X_v \perp\!\!\!\perp X_{V \backslash [v]} | X_{nbrs(v)}$.

A distribution P on V is called *Markov over* g if $X_A \perp\!\!\!\perp X_B | (X_{A\cap B})$ for any decomposition (A, B) of g.

Two distributions Q over X_A and R over X_B are *consistent* if they both yield the same distribution over $X_A \cap X_B$.

A probability measure P on $\mathfrak{X}_V$ is said to *factorise* according to $g = (V, E)$ if for all complete subsets $A \subseteq V$ there exist non-negative functions ψ_A that depend on x_V through x_A only, and there exists a product measure $\mu = \bigotimes_{v\in V} \mu_v$ on $\mathfrak{X}_V$, such that P has density f with respect to μ where f has the form $f(x_V) = \prod_{A \text{ complete}} \psi_A(x_V)$

The next two propositions show that globally Markov distributions over g are exactly the distributions that factorise according to g. Thus every member of a statistical graphical model $\mathfrak{g}$ will factorise according to g.

Proposition 2.6.1 *(Hammersley & Clifford (1971)) A probability distribution P with positive and continuous density f with respect to a product measure μ satisfies the pairwise Markov property with respect to an undirected graph g if and only if it factorises according to g.*

Proposition 2.6.2 *(Proposition 3.8, p. 35, Lauritzen (1996)). For any undirected graph $g = (V, E)$ and any probability distribution P on $\mathfrak{X}_V$, if P factorises according to g, then P is globally Markov with respect to g, which implies the local Markov property, which implies the pairwise Markov property.*

The next lemma guarantees that a globally Markov distribution can be constructed recursively from a set of consistent distributions. Hence decomposable graphical models are completely characterised by sets of consistent clique marginal distributions, and provide a powerful tool for specifying complex multivariate distributions from a set of simple component distributions on the cliques.

Lemma 2.6.3 *(Lemma 2.5, p. 1277, Dawid & Lauritzen (1993)). If distributions Q over X_A and R over X_B are consistent, then there exists a unique distribution P over $X_{A\cup B}$ such that $P_A = Q, P_B = R$, and $X_A \perp\!\!\!\perp X_B | X_{A\cap B} [P]$.*

We now illustrate the construction based on Lemma 2.6.3. Following Dawid & Lauritzen (1993), define the *Markov combination* P of consistent distributions Q over X_A and R over X_B as the unique distribution over $X_{A\cup B}$ such that $P_A = Q, P_B = R$, and $X_A \perp\!\!\!\perp X_B | X_{A\cap B} [P]$. We write $P = Q \star R$.

Let $P = Q \star R$ be the Markov combination of Q and R over $X_{A\cup B}$ with density p. Then using the rules of probability we have

$$\begin{aligned} p_{A\cup B}(x_{A\cup B}) &= p(x_{A\cup B}|x_{A\cap B})p(x_{A\cap B}) \\ &= p(x_A|x_{A\cap B})p(x_B|x_{A\cap B}) \text{ since } X_A \perp\!\!\!\perp X_B|X_{A\cap B} \\ &= \frac{p(x_A, x_{A\cap B})}{p(x_{A\cap B})}\frac{p(x_B, x_{A\cap B})p(x_{A\cap B})}{p(x_{A\cap B})} \text{ by definition of conditional densities} \\ &= \frac{p_A(x_A)p_B(x_B)}{p_{A\cap B}(x_{A\cap B})} \text{ since } A\cap B \subseteq A \text{ and } A\cap B \subseteq B. \end{aligned} \tag{2.11}$$

Since $P = Q \star R$, then if Q, R have densities q, r, respectively, we can substitute these into the above and get

$$\begin{aligned} p(x_{A\cup B}) &= \frac{q(x_A)r(x_B)}{p_{A\cap B}(x_{A\cap B})} \\ &= \frac{q(x_A)r(x_B)}{q_{A\cap B}(x_{A\cap B})} \\ &= \frac{q(x_A)r(x_B)}{r_{A\cap B}(x_{A\cap B})}, \end{aligned} \tag{2.12}$$

since by consistency, all densities agree on $x_{A\cap B}$.

We now illustrate how to generalise this procedure to construct globally Markov distributions from any set of pairwise consistent smaller dimensional distributions, so long as the sample spaces $\{\mathfrak{X}_{A_i} : i = 1, \ldots, k\}$ of the consistent distributions $\{P_{A_i} : i = 1, \ldots, k\}$ can be ordered in such a way that for every $i > 1$, there exists a $j < i$ such that $\mathfrak{X}_{A_i\cap(\cup_{t=1}^{i-1}A_t)} = \mathfrak{X}_{A_i\cap A_j}$. It is self evident that when the marginal specifications of P_V are consistent, then only those for maximal subsets (i.e. cliques) are required. Hence the above condition on the sample spaces is exactly the condition that guarantees the existence of a perfect sequence of the A_i. That is (by Proposition 2.2.1), that the A_i are simply the cliques of a decomposable graph.

Let $g = (V, E)$ be a decomposable graph. Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be a perfect sequence of cliques of g with separators $\mathcal{S} = \{S_2, \ldots, S_k\}$. Assume that P_V is globally Markov with respect to g, and has a pair of consistent marginal densities p_{C_1}, p_{C_2}. Then

$$\begin{aligned} p_{C_2\cup C_1}(x_{C_2\cup C_1}) &= p_{C_2}(x_{C_2})p_{C_1}(x_{C_1})/p_{C_2\cap C_1}(x_{C_2\cap C_1}) \text{ by (2.12)} \\ &= p_{C_2}(x_{C_2})p_{C_1}(x_{C_1})/p_{S_2}(x_{S_2}). \end{aligned} \tag{2.13}$$

Since g is decomposable, for every $i > 1$, each $S_i \in \mathcal{S}$ can be written as $S_i = C_i \cap C_j$ for some $j < i$. This fact together with (2.13) motivates the following construction.

Let $\mathcal{Q} = \{Q_{C_i} : 1 \leq i \leq k\}$ be any set of pairwise consistent clique distributions, where each Q_{C_i} is a distribution of X_{C_i} with associated density q_{C_i}. Define $P_{H_1} = Q_{H_1} = Q_{C_1}, P_{H_2} = P_{C_1 \cup C_2} = P_{H_1} \star Q_{C_2}$, and use the obvious notation for associated densities. Since elements of $\mathcal{Q}$ are pairwise consistent, $q_{C_1}(x_{C_1 \cap C_2}) = q_{C_2}(x_{C_1 \cap C_2})$. Since $H_1 = C_1$, and $S_2 = H_1 \cap C_2$, then $q_{H_1}(x_{H_1 \cap C_2}) = q_{C_2}(x_{H_1 \cap C_2})$. Therefore we have

$$\begin{aligned} p_{H_2}(x_{H_2}) &= p_{H_1}(x_{H_1}) q_{C_2}(x_{C_2}) / q_{H_1 \cap C_2}(x_{H_1 \cap C_2}) \\ &= q_{C_1}(x_{C_1}) q_{C_2}(x_{C_2}) / q_{S_2}(x_{S_2}). \end{aligned} \tag{2.14}$$

Similarly define $P_{H_3} = P_{C_1 \cup C_2 \cup C_3} = P_{H_2} \star Q_{C_3}$. Again, by consistency and the fact that $S_i \subset C_j$ for some $j < i$, it follows that

$$\begin{aligned} p_{H_3}(x_{H_3}) &= p_{H_2}(x_{H_2}) q_{C_3}(x_{C_3}) / q_{S_3}(x_{S_3}) \text{ by (2.14)} \\ &= q_{C_1}(x_{C_1}) q_{C_2}(x_{C_2}) q_{C_3}(x_{C_3}) / q_{S_2}(x_{S_2}) q_{S_3}(x_{S_3}). \end{aligned} \tag{2.15}$$

Giving one more explicit example we have $P_{H_4} = P_{H_3} \star Q_{C_4}$, and so

$$\begin{aligned} p_{H_4}(x_{H_4}) &= p_{H_3}(x_{H_3}) q_{C_4}(x_{C_4}) / q_{S_4}(x_{S_4}) \text{ by (2.15)} \\ &= q_{C_1}(x_{C_1}) q_{C_2}(x_{C_2}) q_{C_3}(x_{C_3}) q_{C_4}(x_{C_4}) / q_{S_2}(x_{S_2}) q_{S_3}(x_{S_3}). \end{aligned} \tag{2.16}$$

In general, since $S_i = H_{i-1} \cap C_i = C_j \cap C_i$ for some $j < i$, and $q_{C_j}(C_j \cap C_i) = q_{C_i}(C_j \cap C_i)$ by consistency, we can recursively define $P_{H_{i+1}} = P_{H_i} \star Q_{C_{i+1}}$. This process can be continued to obtain $p_V(x_V) = p_{H_k}(x_V) = \prod_{i=1}^k q_{C_i}(x_{C_i}) / \prod_{i=2}^k q_{S_i}(x_{S_i})$.

This process can clearly be used to construct a decomposable globally Markov distribution Q_V from any set of consistent clique distributions $\mathcal{Q}$. On the other hand, because the marginal distributions of a joint distribution must be consistent, then for every member P of a decomposable statistical graphical model $\mathfrak{g}$, the clique marginal distributions $P_{C_i}, i = 1, \ldots, k$ are a set of pairwise consistent clique marginals. That is, every decomposable globally Markov distribution can be constructed from some set $\mathcal{Q}$.

Suppose the above procedure is well defined. Then, since every member of $\mathfrak{g}$ can be obtained from some set $\mathcal{Q}$, then any member of a statistical family $\mathfrak{g}$ can be

characterised by its clique marginal distributions, which must be consistent as they are the marginals of the joint p_V. On the other hand, sets of pairwise consistent clique distributions of complete random variables X_{C_i} can be used as 'building blocks' to construct complex larger dimensional distributions, and these distributions will be characterised by the smaller dimensional P_{C_i}.

The next lemma guarantees that the procedure is well defined. Thus every Markov (or hyper Markov) distribution is the recursive Markov combination of a set of consistent clique marginal distributions for a perfect sequence of cliques of some decomposable graph.

Lemma 2.6.4 *(Theorem 2.6, p. 1278, Dawid & Lauritzen (1993)). Let $\mathcal{C}$ and $\mathcal{S}$ be the cliques and separators of a decomposable graph $g = (V, E)$. Let $\{P_C : C \in \mathcal{C}\}$ be a set of consistent clique distributions, with corresponding densities $\{p_C : C \in \mathcal{C}\}$.*

Then the joint density p given by

$$p(x) = \frac{\prod_{C \in \mathcal{C}} p_C(x_C)}{\prod_{S \in \mathcal{S}} p_S(x_S)} \tag{2.17}$$

is the density of the unique Markov distribution over g having the given consistent distributions as its clique marginals.

2.7 Applicability of decomposable models

Using the graphical concept of separation to represent conditional independencies is a valid methodology for inference in any Markov distribution. Therefore the theory of graphs can be applied whenever it is reasonable to assume that a statistical graphical model (i.e. a set of globally Markov distributions) is a subset of the family of distributions that you want to model. Dawid & Lauritzen (1993) construct a range of decomposable Markov subfamily distributions, and give them the prefix 'hyper' to denote that they occur as distributions over the *parameters* of their respective sampling distributions, when it is assumed that the sampling distributions themselves are a decomposable Markov family. For example, they construct the hyper multinomial as the distribution of n times the maximum likelihood estimator of the parameter $p = p_1, \ldots, p_k$ in the multinomial decomposable sampling distribution for $\theta \sim Multinomial(n, p)$; the hyper Wishart as the distribution of the maximum

likelihood estimate of the parameter Σ in the Gaussian decomposable sampling distribution for $y \sim N(0, \Sigma)$; and the hyper inverse Wishart as the conjugate prior distribution for Σ for the same likeihood $y \sim N(0, \Sigma)$. These occur in connection with the analysis of log-linear and covariance selection models for the case when the data are assumed to come from a decomposable distribution.

Chapter 4 considers in detail the hyper Wishart and hyper inverse Wishart distributions and their application in Bayesian covariance selection models. Their use in posterior inference using MCMC and efficient estimation of $E(\Omega|y)$ is analysed in Section 4.8. Section 4.14, Section 5.4 and Section 6.4 give empirical justification for the use of Gaussian decomposable graphical models in covariance selection and general multivariate regression. Based on the simulation evidence reported in this thesis, and the evidence of the datasets considered in Chapters 5 and 6, the decomposable methodology performs comparably to the more general methodologies, even if the actual sampling distribution is not decomposable. In particular, even though the space of decomposable graphs is increasingly sparse with p, (see in Lemma 8.3.1), in the simulations presented in Chapter 4.14, the decomposable covariance estimate appears to be just as 'close' to the true value as the estimate obtained by the more general methodology. It appears that there could be a decomposable distribution sufficiently close to any nondecomposable distribution, and so a decomposable estimate of the covariance may be almost as close to the true value as a nondecomposable estimate, even when the true covariance is not decomposable.

2.8 Using the graphical structure for inference

This section illustrates how to use graph-theoretic concepts to determine whether a minimal factorisation exists. It is the recursive characterisation of V in terms of the sequence $A_i, i = 1, \ldots, k$ and the condition that $X_{R_{i+1}} \perp\!\!\!\perp X_{H_i \setminus S_{i+1}} | X_{S_{i+1}}$ that guarantees the existence of the minimal factorisation, and which makes sequential sampling schemes possible. This recursive characterisation of V is equivalent to the requirement that there exists a perfect sequence of subsets of V. By definition, the index sets A_i of the complete maximal vectors X_{A_i} are exactly the cliques of $g(\Omega)$. So if P_V is globally Markov with respect to $g(\Omega)$, and $g(\Omega)$ is decomposable, then a minimal factorisation exists and we can find the factorisation by finding the cliques

of $g(\Omega)$.

For any matrix M, denote by $M_{A,B}$ the submatrix $M_{A,B}$ of M defined as $M_{A,B} = (M_{a,b} : a \in A, b \in B)$. We write $M_{A,B} = [0]$ to denote that $M_{a,b} = 0$ for all $(a,b) \in A \times B$. Define $M_{A,B}$ to be *full* if $M_{a,b} \neq 0$ for all $(a,b) \in A \times B$. For $A, B \subset V$, the covariance matrix of X_A and X_B is therefore given by $\Sigma_{A,B}$.

The subvectors X_{R_i}, X_{S_i} are complete because the vectors X_{A_i} are complete. Hence the corresponding submatrices $\Omega_{A_i,A_i}, \Omega_{R_i,R_i}, \Omega_{S_i,S_i}$ are full. The equivalence of (i) and (iii) in Proposition 2.3.1 asserts that $\Omega_{R_i,H_{i-1}\backslash S_i} = [0]$. Hence the recursive definitions $R_i = A_i \backslash H_{i-1}$ and $S_i = A_i \cap (\cup_{k=1}^{i-1} A_k)$ can be used to define an ordering $C_1, R_2, \ldots, R_k$ of the index set V which results in Ω having the structure illustrated in Figure 2.1.

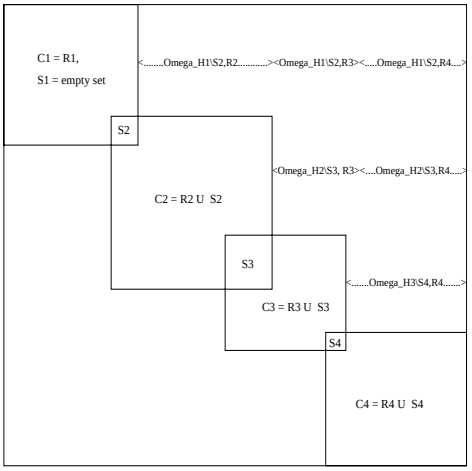


Figure 2.1: Inverse covariance for a decomposable distribution

Ω is diagonal in the special case where $X_v, v \in V$ are mutually independent. This case is referred to as *marginal mutual independence* and it is obvious from Ω that a minimal factorisation exists, and the form of the minimal factorisation. This is not true of the more general cases. First consider the next most general case where X_V can be written as a disjoint union of maximally complete $X_{A_i}, i = 1, \ldots, k$ which are mutually independent of each other, and call this case *disjoint block independence*. If the variables are ordered correctly, then disjoint block independence can be inferred from the inverse covariance. But if the ordering is not correct, then this is not true. This is illustrated by comparing *greyscale* representations of generic inverse

covariances, where the greyscale is created by putting a white square in the ijth position of the figure if $\Omega_{ij} = 0$ or $i = j$, and a black square otherwise. A white square is used for $i = j$ so that the greyscale of Ω is identical to the greyscale of the adjacency matrix of $g(\Omega)$.

Figure 2.3 makes disjoint block independence obvious. Figure 2.2 is the greyscale of the same inverse covariance but now the variable ordering is permuted: in this figure neither the required ordering nor the subsets of variables is obvious. The pictorial representation is independent of variable ordering when labels on the vertices are ignored. Figures 2.5 and 2.4 give the pictorial representations of Figure 2.3 and Figure 2.2 respectively. They are identical if the labels showing the ordering of the index set in the corresponding greyscales are deleted.

The argument is made stronger by considering distributions which do not satisfy marginal mutual or disjoint block independence. In Figure 2.7 it is arguably discernable from Ω that the distribution is decomposable, but here the vertices of the index set are ordered correctly. Figure 2.6 is the greyscale of the same inverse covariance, but now the variable ordering has been permuted: in this figure neither the required ordering nor the subsets A_i are obvious. On the other hand, the structure of the pictorial representation is independent of ordering. Figures 2.9 and 2.8 are the pictorial representations of Figure 2.7 and Figure 2.6 respectively. They are identical if the labels showing the ordering of the vertices in the corresponding greyscales are deleted.

The pictorial representation faciliates the inference of when decomposability holds, which is not true of the inverse covariance. The greyscale in Figure 2.10 is nondecomposable. It is obtained by adding a single illegal edge to the decomposable greyscale given in Figure 2.6. It is very difficult to distinguish these greyscales at a glance. Figure 2.11 is the nondecomposable graph of Figure 2.8 obtained by adding the same single illegal edge. It is very easy to distinguish these pictorial representations at a glance, and in particular, to determine the illegal 7-cycle in Figure 2.11.

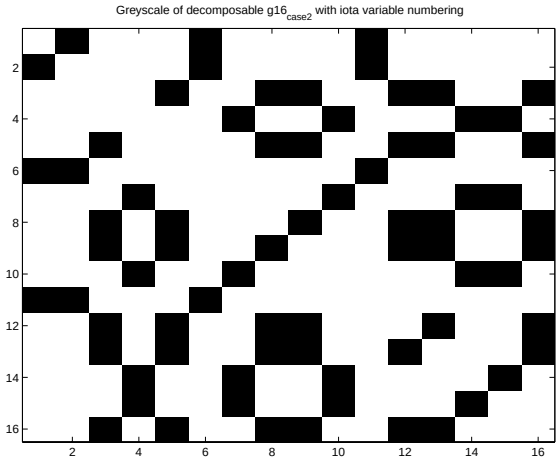


Figure 2.2: 16-dimensional example of block independence, a special case of Sundberg's criterion. A perfect numbering of variables exists, but the order of the sequence of variables chosen for the covariance is not a perfect numbering.

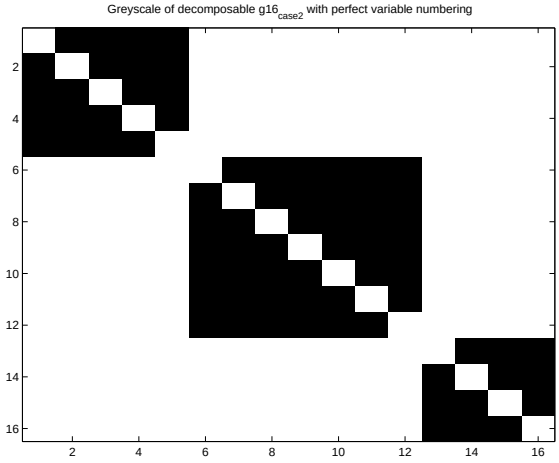


Figure 2.3: The same inverse covariance as Figure 2.2, but with the covariance ordering chosen to match a perfect numbering.

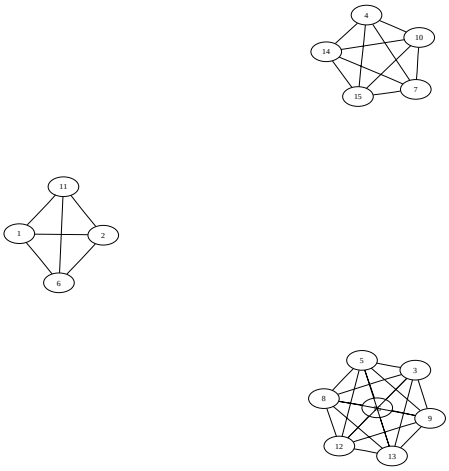


Figure 2.4: 16-dimensional pictorial representation of Figure 2.2

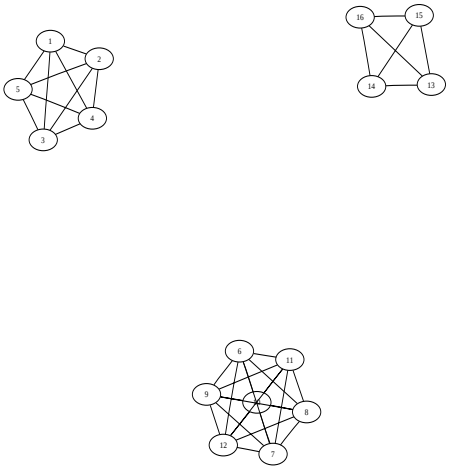


Figure 2.5: 16-dimensional pictorial representation of Figure 2.3

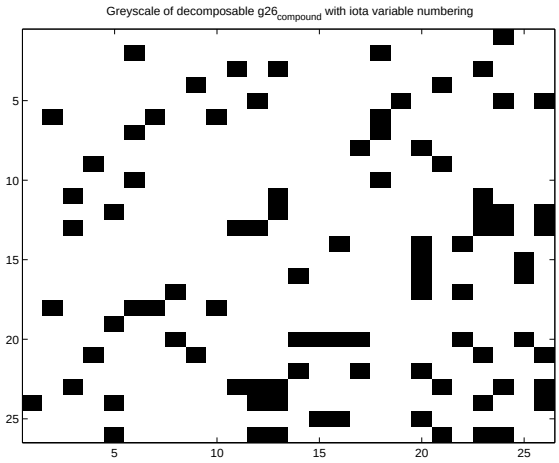


Figure 2.6: 26-dimensional Sundberg's criterion example where a perfect numbering of variables exists, but the order of the sequence of variables chosen for the covariance is not a perfect numbering.

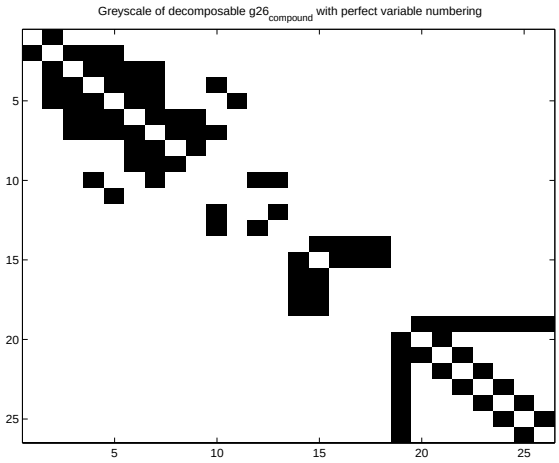


Figure 2.7: The same inverse covariance as Figure 2.6, but with the covariance ordering chosen to match a perfect numbering.

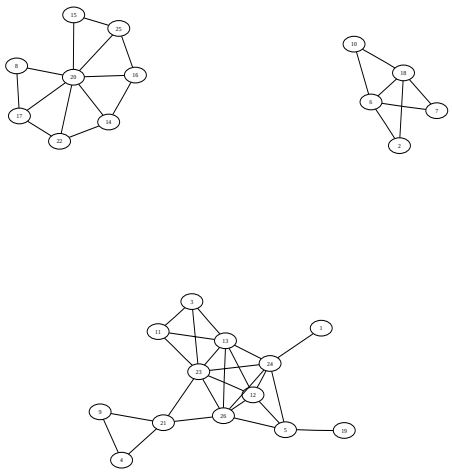


Figure 2.8: 26-dimensional pictorial representation of Figure 2.6

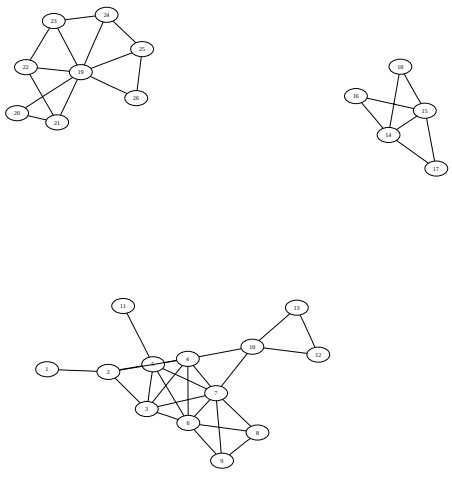


Figure 2.9: 26-dimensional pictorial representation of Figure 2.7

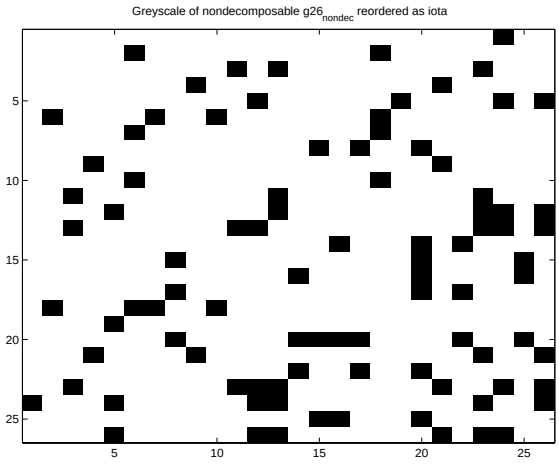


Figure 2.10: Greyscale of nondecomposable graph got by adding a single illegal edge

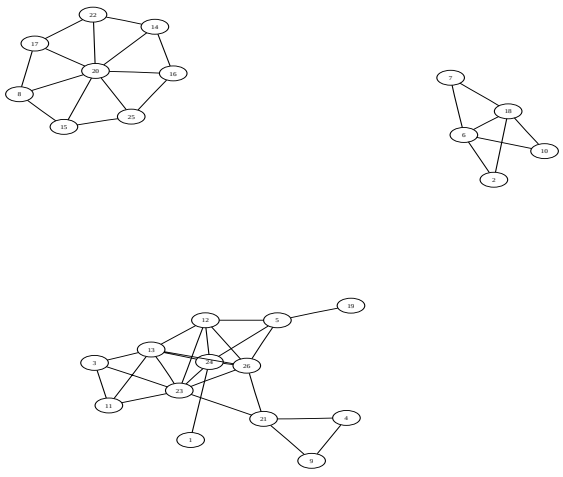


Figure 2.11: Pictorial representation of same nondecomposable graph as Figure 2.10

2.9 Historical motivation for decomposable distributions and Sundberg's criteria

The aim is find a minimal factorisation of p_V. If there exists a triple A_1, A_2, S of disjoint subsets of V with union V such that $X_{A_1} \perp\!\!\!\perp X_{A_2} | X_S$, then p_V can be factorised as

$$\begin{aligned} p_V(x_V) &= p(x_{A_1}|x_{A_2}, x_S)p(x_{A_2}|x_S)p(x_S) \\ &= p(x_{A_1}|x_S)p(x_{A_2}|x_S)p(x_S) \quad \text{since} \quad X_{A_1} \perp\!\!\!\perp X_{A_2}|X_S. \end{aligned} \tag{2.18}$$

If no such triple exists, then the following characterisation is a more general recursive 'decomposition' of the index set V into sets $A_i, i = 1, \ldots, k$, with consequent minimal factorisation of p_V. We call this characterisation *Sundberg's criterion*, to acknowledge that it is equivalent to the criterion of decomposability given by Sundberg (1975).

Sundberg's criterion There exists an ordered sequence of maximally complete subvectors $X_{A_1}, \ldots, X_{A_k}$ with union X_V, such that each X_{A_i} can be written as a disjoint union of 'new' elements $X_{R_i}, R_i \nsubseteq A_j, j < i$ and 'old' elements $X_{S_i}, S_i \subset A_j, j < i$, and for every $i > 1$, $X_{R_i} \perp\!\!\!\perp X_{(\cup_{t=1}^{i-1} A_t)\setminus S_i} | X_{S_i}$.■

We say that any distribution satisfying Sundberg's criterion is *decomposable*.

If we let $H_{i-1} = \cup_{t=1}^{i-1} A_t$, then $S_i = A_i \cap H_{i-1}$ and $R_i = A_i \setminus H_{i-1}$. Since $S_i \subset A_j$, then $S_i = A_i \cap A_j$. Hence $X_{R_i} \perp\!\!\!\perp X_{H_{i-1}\setminus S_i} | X_{S_i}$, and the 'new', or 'residual' random variables in the subvector X_{R_i} of X_{A_i} can be made independent of the total 'history' of random variables in $X_{H_{i-1}}$ by conditioning on X_{S_i} for $S_i = A_i \cap H_{i-1}$.

Thus Sundberg's criterion leads to a minimal factorisation. If the A_i are all disjoint and $\perp\!\!\!\perp \{X_{A_1}, \ldots, X_{A_k}\} [P_V]$, then we have the special case where every conditioning set x_S in the factorisation is empty. The Bayesian methodology for sequentially generating the random variables in X_V in an MCMC scheme that minimises the conditioning required at each step is obvious from the above factorisation. First sample $X_{H_1} = X_{A_1}$. Continue to sample each $X_{R_{i+1}}$ conditional on the previously generated values $x_{S_{i+1}}$ of $X_{S_{i+1}}$. The values $x_{S_{i+1}}$ are known because $S_{i+1} \subset H_i$ and the values x_{H_i} have already been sampled. This scheme maximises the number of variables sampled at each step, as each X_{R_i} is the maximal subvector of the

maximally complete X_{C_i} that has not yet been sampled. Equivalently, the factorisation has the fewest possible factors. On the other hand, Section 2.6 shows that the conditions of Sundberg's criterion guarantee that a well defined joint distribution for X_V with desirable independence properties can be constructed from any set $\{P_{A_i}, i = 1 \dots k\}$ of simple lower-dimensional distributions of the subvectors X_{A_i} that agree on their pairwise common sample spaces.

2.10 Appendix to Chapter 2

The next lemma shows that $R_i \overset{g}{\perp} H_{i-1} | S_i$, is true in every decomposable graph. This is a well known result which is proved here for completeness because the best known reference (Lauritzen, 1996,Lemma 2.11, p. 15) is for the special case $R_i \overset{g_{H_i}}{\perp} H_{i-1} | S_i$, and we need to guarantee separation in g, not just g_{H_i}.

Lemma 2.10.1 *Let $g = (V, E)$ be a decomposable graph with perfect sequence of cliques $C_1, \dots, C_k$. For $i \geq 2$, define $R_i = C_i \backslash H_{i-1}$ and $S_i = C_i \cap H_{i-1}$ where $H_j = \cup_{t=1}^{j} C_t$. Then $R_i \overset{g}{\perp} H_{i-1} \backslash S_i | S_i$.*

Proof. We need to show that every path $(r, \dots, h)$, where $r \in R_i, h \in H_{i-1} \backslash S_i$, includes at least one $s \in S_i$, or, equivalently, that r and h are not connected in $g_{V \backslash S_i}$. Without loss of generality, assume that $(r, \dots, h)$ is the shortest path in $g_{V \backslash S_i}$ connecting R_i and H_{i-1}, and note that any shortest path contains at most one edge from any clique.

We first show that only paths in g_{H_i} need to be considered. Let $\mathcal{T}$ be the graph with vertices $C_1, \dots, C_k$, and with edges obtained by successively choosing for each $i > 2$, the smallest $j < i$ for each i such that $S_i \subset C_j$ and then letting $C_i \sim C_j$. By Proposition 2.8, p. 26, Lauritzen (1996), $\mathcal{T}$ is a junction tree and, in particular, contains no cycles.

If the path $(r, \dots, h)$ includes any vertex $v \in C_t \backslash C_i, t > i$, then $\mathcal{T}$ includes a cycle $(C_j, C_i, \dots, C_t, \dots, C_j)$, contradicting the fact that it is a tree.

So assume that $(r, \dots, r', h', \dots, h) \in g_{H_i}$ where $(r', h') \in g_{H_i \backslash S_i}$ is an edge between a vertex $r' \in R_i$ and a vertex $h' \in H_{i-1} \backslash S_i$. By definition of R_i and H_{i-1}, then $r \notin C_t$ for any $t < i$ and $h' \notin C_i$. We have already ruled out the case that $r', h' \in C_t$. So there is no clique containing the pair r', h', hence there is no such edge (r', h').

Therefore the path $(r, \ldots, r', h', \ldots, h)$ does not exist. Hence S_i is an (r, h)-separator of all pairs $r \in R_i, h \in H_{i-1} \backslash S_i$ and the result is proved. ■

Chapter 3

Decomposable graphical models

3.1 Introduction

The pictorial representation of a graphical model enables us to discern independence and conditional independence properties easily. Chapter 2 shows that the density of X_V can be factored in terms of lower dimensional components to make it more tractable. This chapter discusses conditions to determine whether a graphical model is decomposable, and, if so, how to determine the sets required for a minimal factorisation. Section 3.3 shows how to transition from one decomposable graph to the next in such a way that the Markov chain generated is irreducible, and includes some new results for determining when a decomposable graphical model remains decomposable when we add an edge. Section 3.5 gives a number of detailed examples to illustrate the graph-theoretic concepts defined in Section 2.2 and discussed in this chapter.

MATLAB algorithms to implement the graph-theoretic procedures are given in in Appendix 8.1, and the equivalent FORTRAN code is given in Appendix 8.4. The code presented in these appendicies is the actual code used to produce the results in this thesis. A working directory of the same is available upon request.

3.2 Equivalent definitions of decomposable graphs

Leimer (1989) shows that a graph g has no unchorded n-cycles, $n \geq 4$, if and only if it is decomposable. Therefore any distribution which is globally Markov with respect to g, where g has no unchorded n-cycles, $n \geq 4$, is decomposable. We show

in Section 3.5 that checking for unchorded n-cycles in the pictorial representation is quite simple, and so provides a valuable methodology for checking decomposability of g.

Determining decomposability by checking that there are no unchorded n-cycles, $n \geq 4$, is equivalent to the algorithm of Tarjan & Yannakakis (1984) which is commonly cited in the literature and is straightforward to program. Furthermore, Tarjan & Yannakakis (1984) outputs an ordering of the variables which is a perfect numbering if the graph is chordal, and aborts otherwise. This perfect numbering can then be used to find a perfect sequence of cliques and the separators, residuals and histories of the sequence, as illustrated in Section 3.5.3. Therefore it is the Tarjan & Yannakakis (1984) algorithm that is used as the basis of the MATLAB algorithm for checking decomposability that is given in Appendix 8.1.

Graphs which have no unchorded n-cycles, $n \geq 4$, are commonly called *triangulated*, and the result of Leimer (1989) is consequently given as 'decomposable if and only if triangulated'. This terminology is misleading as illustrated by Figure 3.3. The pictorial representation looks 'triangular', yet we show in Section 3.5 that it is not chordal, and therefore not decomposable by Leimer (1989). That is, it is comprised entirely of 3-edged segments, but it is *not* triangulated. It is easy to create many such nondecomposable graphs with a triangular appearance using similar amalgamations of 3-edged segments. Therefore we prefer to use the terminology chordal or decomposable rather than triangulated.

3.3 Legal edge deletion and addition

This section discusses the graph-theoretic concepts required to define an irreducible Markov chain on the decomposable graph space. Consider the state space $\mathfrak{g}$ of all decomposable graphs with vertices V. Let $g^{[1]}, g^{[2]}, \ldots$ be a Markov chain of states from $\mathfrak{g}$ where the one step transition from state $g^{[i]}$ to state $g^{[i+1]}$ is conditional on $g^{[i]}$ and is given by adding or deleting a single edge in $g^{[i]}$ such that the resulting graph remains decomposable.

We generate the Markov chain of graphs using a Metropolis Hastings Markov chain Monte Carlo (MH MCMC) algorithm. In order to ensure convergence to the correct distribution, it is necessary to ensure that the chain is irreducible, and

that each graph sampled is decomposable. That is, given any graph in the space, there must be a positive probability of sampling any other decomposable graph. Lemma 3.3.1 ensures that this chain is irreducible.

Lemma 3.3.1 *(Lemma 5, Frydenberg & Lauritzen (1989)) Let $g = (V, E)$ be decomposable and let $g' = (V', E')$ be a decomposable subgraph of g such that $|E/E'| = k$. Then there exists an increasing sequence $g' = g_0 \subset g_1 \subset \ldots \subset g_k = g$ of decomposable graphs that differ by exactly one edge.*

If changing a single edge in a decomposable graph results in a decomposable graph, the edge change will be called *legal*. In order to make the transition from $g^{[i]}$ to another graph within $\mathfrak{g}$, we must either test the legality of a given edge change or else check whether a proposed graph is decomposable. We will see that characterising legal changes is more computationally efficient than checking that g is decomposable, and so the characterisation of legal edge changes is fundamental to the Metropolis Hastings Markov Chain Monte Carlo methodology used in this thesis. Therefore a number of examples of legal and illegal edge additions are given in Subsection 3.5.3. These examples illustrate how to check if a given edge change is legal, and make the proof of Lemma 3.3.4 easier to understand.

Lemma 3.3.2 is a characterisation of legal deletions from a decomposable graph originally given by Frydenberg & Lauritzen (1989). Lemma 3 on p. 551 of Frydenberg & Lauritzen (1989) states without proof that the deletion of an edge that is contained in two or more cliques results in a chordless 4-cycle, and conversely. For completeness, we prove this.

Lemma 3.3.2 *(Frydenberg & Lauritzen, 1989,Lemma 3). Let $g = (V, E)$ be decomposable, and $e = (a, b)$. Then $g - e$ is decomposable if and only if e is a member of only one clique.*

Proof. Assume that $e = (a, b) \in C_i$ and $e \in C_j$ for some pair of cliques with $C_i \neq C_j$. We must show that there exists an unchorded n-cycle, $n \geq 4$ in $g - e$. Since $C_i \neq C_j$, there exists $x \in C_i$ such that $x \notin C_j$. As $x \notin C_j$, then by completeness of cliques there exists at least one $y \in C_j$ such that $x \nsim y$. Consider the cycle (a, x, b, y, a) in g. Since $x \nsim y$, we know that (a, b) is the only chord on this cycle. Thus deleting e results in an unchorded 4-cycle in $g - e$.

Now assume that $e = (a, b) \in C$ and $e \notin C_j$ for any $C_j \neq C$. We must show that every n-cycle, $n \geq 4$ in $g - e$, contains a chord. Let $(a, x, \ldots, b, \ldots, y, a)$ be any n-cycle in $g - e$, $n \geq 4$. Since (a, b) is in only one clique C, then C must contain all the neighbours of a. In particular, both x and y belong to C. Since C is complete, we have $x \sim y$ so (x, y) is a chord on this cycle. ■

Adding edges is harder. Consider a pair of vertices u, v such that $u \nsim v$. For any edge $e = (u, v)$ to be an illegal addition to g, an unchorded n-cycle $(u, v_0, \ldots, v_k, v, u)$, $n \geq 4$ must be created by making u and v adjacent in $g + e$. There must therefore exist a path $(u, v_0, \ldots, v_k, v)$ in g such that u is not adjacent to any $v_i, i \neq 0$, and v is not adjacent to any $v_i, i \neq k$ (as any one of these adjacencies creates a chord on the cycle). This gives the following simple necessary test for legal edge additions: there must exist a path of length 2 between the vertices.

Lemma 3.3.3 *An edge addition (v_i, v_j) to a connected decomposable graph g is illegal if its adjacency matrix satisfies $\sum_{r=1}^{p} G_{ir}G_{rj} = 0$, or equivalently, $d_g(v_i, v_j) \geq 3$.*

Proof. It is a basic fact in graph theory that if A is a $p \times p$ adjacency matrix, and $A^k = (A^k_{ij})$, then $A^k_{ij} = \sum_{r=1}^{p} A^{k-1}_{ir} A_{rj}$ equals the number of paths from vertex v_i to vertex v_j of length k. (See, for example, Theorem 4.14, p. 223, Kalmanson (1986).) If $\sum_{r=1}^{p} G_{ir}G_{rj} = 0$, there is no length 2 path (v_i, v_0, v_j) including a vertex v_0 adjacent to both v_i and v_j. That is, $d_g(v_i, v_j) \geq 3$. Consider any shortest path $(v_i, v_0, \ldots, v_j)$ in g and consequent shortest cycle $(v_i, v_0, \ldots, v_j, v_i)$ in $g + e$. This is an unchorded n-cycle, $n \geq 4$ because any chord would give a shorter path from v_i to v_j in g, which is a contradiction. ■

The graph we refer to as $g8$ in Figure 3.13 illustrates that Lemma 3.3.3 is not a sufficient condition for legal additions. There is a length 2 path connecting vertices 7 and 1, but if $e = (7, 1)$ is added, an unchorded 7-cycle $(1, 2, 3, 4, 5, 6, 7, 1)$ results in $g8 + e$, so e is an illegal addition.

Lemma 3.3.3 is a useful sufficient test for illegality of an edge addition proposal in an MCMC sampler. It can be used to decrease the computational effort required when proposing edge additions to a graph with a relatively large number of edges, because most decomposable graphs have approximately half the $2^{p(p-1)/2}$ total possible number of edges (see Section 7.5). That is, when the number of edges is relatively large in comparison to the total possible number of edges, there are relatively fewer

decomposable than nondecomposable graphs having more edges than the current graph. So when the current decomposable graph has a relatively large number of edges, there is a priori more chance of randomly proposing an illegal rather than legal edge addition.

Lemma 3.3.4 gives a simpler, though equivalent, characterisation of legal additions to Theorem 2 of Giudici & Green (1999). It can be discerned directly from the graph without building a junction tree (to be defined in Section 3.4). Lemma 3.3.4 is therefore easier to verify quickly from the pictorial representation than Theorem 2 of Giudici & Green (1999). We also show that Lemma 3.3.4 is easier to program (inefficiently), and that it can be programmed efficiently.

Lemma 3.3.4 *Let g be a decomposable graph in which vertices u, v are not adjacent. Then $e = (u, v)$ is a legal addition to g if and only if there exists a pair of cliques C_u, C_v such that $u \in C_u$, $v \in C_v$ and their intersection $C_u \cap C_v$ is a (u, v)-separator.*

Note that if u and v are in different connected components, then intersections of $C_u \cap C_v = \emptyset$ for every pair C_u, C_v containing u and v respectively. But if u and v are not connected in g, then they are separated trivially by the empty set. Hence the characterisation of Lemma 3.3.4 subsumes both the connected and the disconnected case.

The following is a stand alone proof. Necessity would be made simpler if we assume Theorem 2, Giudici & Green (1999), since if the Giudici & Green (1999) Theorem 2 condition holds, then there must exist cliques C_u and C_v such that $C_u \cap C_v$ is a (u, v)-separator. Sufficiency for Lemma 3.3.4 is a lot simpler than for Theorem 2, Giudici & Green (1999).

Proof. Assume throughout that $u \nsim v$ in g. The disconnected case is trivial: if there is no path connecting u and v in g, no cycle can be created in $g + e$ by making $u \sim v$, and all intersections are empty. In this case, $\emptyset$ is trivially a (u, v)-separator so the characterisation holds.

In the rest of the chapter we use the notation C_u, C_v to indicate a pair of cliques of g containing the vertices u, and v respectively.

For g connected, consider 3 cases. ***Case 1: First assume that there exists a pair of cliques C_u, C_v such that $S = C_u \cap C_v$ is a (u, v)-separator.*** Refer to Figure 3.1. We need to show that every n-cycle, $n \geq 4$, in $g + e$ contains a chord.

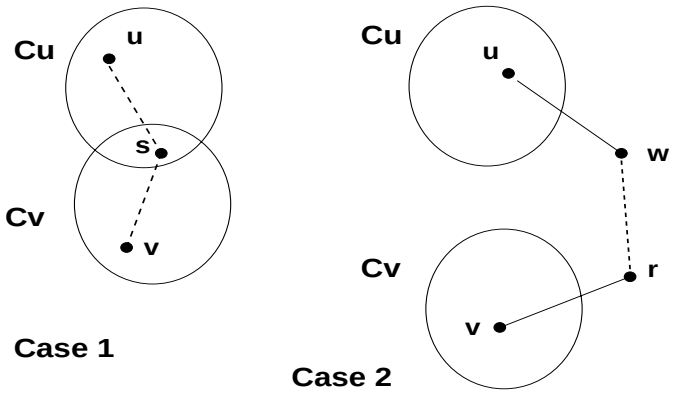


Figure 3.1: Illustration of Case 3

Since S is a separator, every path $(u, \ldots, v)$ in g contains at least one $s \in S$. There are three possibilities:

1. *The path is of form* $(u, \ldots, s, \ldots, v)$ *with* $i \geq 1$ *vertices between* u *and* s*, and* $j \geq 1$ *vertices between* s *and* v*:* Since C_u is complete and $s \in C_u$, $u \sim s$. Therefore (u, s) is a chord on the cycle $(u, \ldots, s, \ldots, v, u)$ in $g + e$.

2. *The path is of form* $(u, s, \ldots, v)$ *with* $j \geq 1$ *vertices between* s *and* v*:* C_v is complete and $s \in C_v$, so $s \sim v$. Therefore (v, s) is a chord on the cycle $(u, s, \ldots, v, u) \in (g + e)$.

3. *The path is of the form* (u, s, v)*:* Since this path comprises only 3 vertices, (u, s, v, u) is a 3-cycle.

Hence there are no unchorded n-cycles, $n \geq 4$, created by adding the edge (u, v) to g in this case.

Case 2: Next assume that every pair of cliques C_u***,*** C_v ***is such that*** $C_u \cap C_v = \emptyset$**.** Refer to Figure 3.1. Consider a shortest path $(u, w, \ldots r, v)$ in g. If $d_g(u, v) \geq 3$, then there exist $w \sim u$ and $r \sim v$ such that $(u, w, \ldots r, v, u)$ is an unchorded n-cycle in $g + e$. Hence it suffices to show that $d_g(u, v) \geq 3$. Assume for a contradiction that $d_g(u, v) = 2$, and consider a shortest path (u, w, v). Since $u \sim w$, there exists a clique C_u containing u and w. Similarly, there exists a clique C_v containing v and w. Therefore $w \in (C_u \cap C_v)$ for this pair of cliques, contradicting the assumption of this case.

Case 3: Finally assume that there exists a pair of cliques C_u***,*** C_v ***such that*** $C_u \cap C_v \neq \emptyset$***, but*** $C_i \cap C_j$ ***is not a separator for all pairs of cliques*** C_i***,*** C_j ***such that*** $u \in C_i$***,*** $v \in C_j$***.*** Let C_u, C_v be any pair of cliques containing u and v respectively. Let S^* be a (minimal) (u, v)-separator containing $C_u \cap C_v$. Such an S^* exists because any (u, v)-separator must contain all the common neighbours of u and v. Since every vertex in C_u is a neighbour of u and similarly for C_v, then S^* must contain $C_u \cap C_v$ for every pair of cliques containing u and v respectively.) Further, since g is decomposable we know that we can take S^* to be complete.

Let $W = C_u \cap C_v$ and consider the graph $g - W$. There are two possibilities. Either $d_{g-W}(u, v) \geq 3$, or $d_{g-W}(u, v) = 2$ (because $e \notin g$, so $u \nsim v$). If $d_{g-W}(u, v) \geq 3$, then there exists a shortest path from u to v in $g - W$ of the form $(u, a_1, \ldots, a_k, s, b_1, \ldots, b_t, v)$ where $s \in S^* \backslash W$ and either $k \geq 1$ or $t \geq 1$. Then adding the edge $e = (u, v)$ gives an unchorded n-cycle, $n \geq 4$, in $g - W + e$. This is also an unchorded n-cycle in $g + e$. So we cannot add the edge e in this case.

Otherwise, there is a path from u to v in $g-W$ of length 2. Refer to Figure 3.2. Let

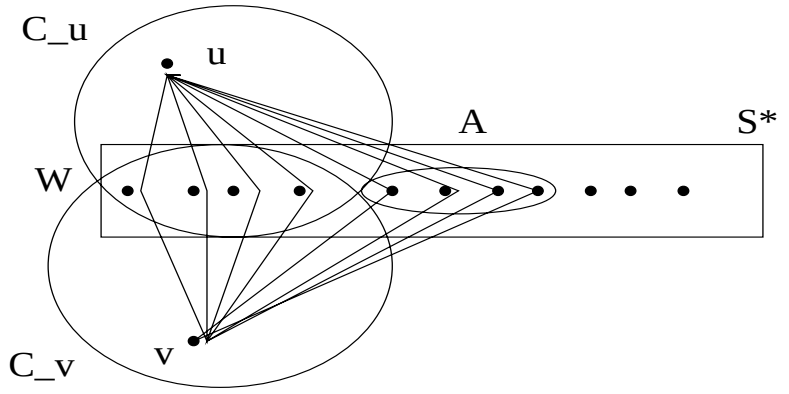


Figure 3.2: Illustration of Case 3

$a_1, \ldots, a_v$ be all the common neighbours of u and v in $g-W$, and let $A = \{a_1, \ldots, a_v\}$. We have already argued that S^* contains all such common neighbours, so $A \subseteq S^*$. Since $u \nsim v$ in g, then the vertices in $W \cup A$ must span a complete graph or else there would exist an unchorded 4-cycle of the form (u, i, v, j, u) for some pair $i, j \in W \cup A$, contradicting that g is chordal. (That $W \cup A$ spans a complete graph also follows since $W \subseteq S^*$ and the fact that S^* is complete because g is chordal.) So all elements of A are joined to each other, all joined to both u and v, and all joined to each

element in $W = C_u \cap C_v$. Hence there exists a maximally complete set (i.e. a clique) $\widehat{C_u}$ such that $\{u\} \cup W \cup A \subseteq \widehat{C_u}$. Similarly, there exists a clique $\widehat{C_v}$ such that $\{v\} \cup W \cup A \subseteq \widehat{C_v}$. Further, $\widehat{C_u} \cap \widehat{C_v} \subseteq S^*$ because any vertex $z \in \widehat{C_u} \cap \widehat{C_v}$ is by construction a common neighbour of both u and v, and so must be in either W or A. Note that by construction, $|\widehat{C_u} \cap \widehat{C_v}| = |C_u \cap C_v| + |A| > |C_u \cap C_v|$ (and the inequality is strict). Because of the strict inequality, $|S^* \backslash (\widehat{C_u} \cap \widehat{C_v})| < |S^* \backslash (C_u \cap C_v)|$. If we now choose $\widehat{C_u}$ and $\widehat{C_v}$ as the pair of cliques to consider, we can repeat this process. By the decreasing size of the sets $|S^* \backslash (\widehat{C_u} \cap \widehat{C_v})|$, this process must terminate for some final pair of cliques $\widehat{C_u}, \widehat{C_v}$, with $u \in \widehat{C_u}$, $v \in \widehat{C_v}$, and $\widehat{C_u} \cap \widehat{C_v} \neq \emptyset$. There are only two eventual outcomes. Either $\widehat{C_u} \cap \widehat{C_v} = S^*$, which contradicts the assumption in this case, or $d_{g-(\widehat{C_u} \cap \widehat{C_v})}(u, v) \geq 3$, in which case the edge (u, v) cannot be added.

Cases 2 and 3 give necessity of the conditions and the lemma is proved. ■

It is easy to program Lemma 3.3.4 when equivalently stated in terms of path matrices, as in Lemma 3.3.6. The program in Section 8.1.13 programs this lemma naively, and is not as computationally efficient as more sophisticated alternatives. First, it has the disadvantage of searching over all pairs of cliques that contain u and v. Second, calculating the path matrix is of complexity $O(p^4)$.

More efficient algorithms can be substituted for the conceptually simpler program once the user is sufficiently confident with the graph-theoretic concepts required to do so. More efficient algorithms for finding the path matrix exist. A more efficient overall approach based on the junction tree of the modified graph $g' = g - (C_u \cap C_v)$, is as follows. Remove $C_u \cap C_v$ from the perfect sequence of the cliques of g, and hence find a perfect sequence containing the cliques g'. Lemma 4.4.4 can then be applied to find a perfect sequence of cliques of g'. Construct the junction tree $\mathcal{T}'$ of g', and check to see if u and v are disconnected in $\mathcal{T}'$.

Note that the condition that $C_u \cap C_v \subsetneq S$ for some separator S is not sufficient. It is shown in the above proof that every such intersection is a subset. The intersection must be equal to a separator. Figure 3.13 illustrats an illegal edge addition ($u = 1, v = 7$) for which $C_u \cap C_v$ is a proper subset of S.

Remark 3.3.5 *Theorem 2 of Giudici & Green (1999) is a significant contribution to the implementation of decomposable graphical models in statistical inference. Prior to their characterisation, it was necessary to use computationally costly algorithms such as the Maximum Cardinality Search of Tarjan & Yannakakis (1984) (complexity*

$\mathcal{O}(|V| + |E|)$) to generate a chain of decomposable graphs by legal edge changes.

Lemma 3.3.6 *Let g be a decomposable graph with cliques $\mathcal{C}$. Let $\mathcal{C}_u = \{C \in \mathcal{C} : u \in C\}$, and $\mathcal{C}_v = \{C \in \mathcal{C} : v \in C\}$. Let $C_u \in \mathcal{C}_u$ and $C_v \in \mathcal{C}_v$ be such that $|C_u \cap C_v|$ is maximal. Then the edge (u, v) is a legal addition to g if and only if the uvth entry of the path matrix of $g - (C_u \cap C_v)$ is zero.*

Proof. This follows directly from the definition of a path matrix, and from the construction of the separator S^* in the proof of Lemma 3.3.4. ■

Lemma 3.3.6 is easy to program. First use *find_all_clique_containing.m* (described in Subsection 8.1.9) to find the sets $\mathcal{C}_u$ and $\mathcal{C}_v$. Next test the size of all pairwise intersections between $\mathcal{C}_u$ and $\mathcal{C}_v$ to find a pair C_u, C_v with maximal intersection. Calculate the adjacency matrix of $g - (C_u \cap C_v)$ by making zero all rows and columns indexed by the elements of $C_u \cap C_v$. Use *reachability_graph.m* (described in Subsection 8.1.5) to find the path matrix of $g - (C_u \cap C_v)$. Then the edge $e = (u, v)$ may be added if and only if the uvth entry of the path matrix is zero.

Excluding junction tree code gives less computationally complicated code than that based on Theorem 2 of Giudici & Green (1999), and explained in Subsection 8.1.12. Compare the code of Subsection 8.1.12 to that of Subsection 8.1.13. There are more than 4 times as many lines of code in the program based on Theorem 2 of Giudici & Green (1999) than the code in Subsection 8.1.13 which is based on Lemma 3.3.6.

Remark 3.3.7 *The characterisations for legality of single edge transitions does not include reference to any particular distribution. Therefore a Markov chain on the state space of decomposable graphs can be generated using these methods and any transition probability. In Section 7.3 we choose a transition prior on the state space that results in an MCMC chain which converges to a distribution which gives equal probability to all graphs of the same number of edges. In Section 4.8 we choose a transition probability that results in an MCMC chain which converges to the posterior distribution of the decomposable state space.*

3.4 Junction trees

We discuss junction trees in this section because they can be used to simplify algorithmic procedures and increase computational efficiency in decomposable models. They also often simplify proofs of decomposable graph-theoretic results, and are used in the characterisation of legal edge additions that is given by Giudici & Green (1999).

Let $\mathcal{C}$ be a collection of subsets of V. Let $\mathcal{T}$ be a tree with vertices that correspond to the elements of $\mathcal{C}$. That is, each vertex in the tree corresponds to a single subset in $\mathcal{C}$. Then $\mathcal{T}$ is a *junction tree* if any intersection $C_i \cap C_j$ (where $C_i, C_j \in \mathcal{C}$) is contained in every vertex on the (necessarily unique) path in $\mathcal{T}$ between C_i and C_j.

Let $\mathcal{C}$ be the cliques of a decomposable graph, and the sequence $C_1, \ldots, C_k$ a perfect sequence of the cliques. Then we know that for all $i > 1$, there is a $j < i$ such that the separator $S_i \subseteq C_j$. In Subsection 3.5.2 we illustrate how every S_i can be obtained as the intersection between at least one pair $C_i \cap C_j$ for some $j < i$. So if a tree is created using a perfect sequence of cliques $C_1, \ldots, C_k$ as the vertices, then that tree will satisfy the definition of a junction tree if the edges are successively chosen as follows. For each $i > 1$, add the edge (C_i, C_j) to one of the $C_j, j < i$ that satisfies $|C_i \cap C_j|$ is maximal. Note that if there are more than one C_j which satisfies, then any of these can be chosen.

Junction trees can be interpreted as perfect sequences with parallel subsequences, where for every $i > 1$, you 'split out' and make 'parallel' the subsequences that have the same intersection set with C_i. These splits occur as the branches in the junction tree at C_i.

3.5 Illustrative examples

The pictorial representation makes concepts intuitively clear, because finding paths amounts to tracing along lines between the symbols used for vertices. This section illustrates, using detailed examples, the definitions of Section 2.2, and the graph-theoretic concepts presented in this chapter.

3.5.1 Representing conditional independence properties

In Figure 2.9 $X_1 \perp\!\!\!\perp \{X_{14}, X_{21}\}$, $X_1 \perp\!\!\!\perp \{X_{14}, X_{21}\} | X_D$ for any subset of variables D, and $X_1 \perp\!\!\!\perp X_B | (X_D \cup X_2)$ for all subsets B and D because 2 is a $(1, v)$-separator for every vertex $v \in V \backslash 2$.

If there exists at least one path between an element of A and an element of B which does not include an element of D, then D does *not* separate A and B, and so $X_A \not\perp\!\!\!\perp X_B | X_D$. For example, Figure 2.9 implies that $X_1 \not\perp\!\!\!\perp (X_B \cup \{X_{10}, X_{14}, X_{21}\}) | X_D$ whenever D excludes the vertex 2, because 2 is on every path between 1 and 10. Note that none of the sets are assumed to be minimal or maximal, and that the graphical representation of conditional independence using graph separation is a general feature of all graphical models, not just decomposable graphs.

It is clear that in a decomposable graph, the minimal separating set between a clique C_i and H_{i-1} is the sequence separator $S_i = C_i \cap H_{i-1}$. Lemma 2.10.1 shows this.

3.5.2 Checking chordality

Consider the graph in Figure 3.3, which we refer to as $g5$. It contains a 5-cycle $(1, 2, 3, 4, 5, 1)$. None of the pairs of vertices which are *not* adjacent on the 5-cycle (i.e. 1 and 3, 1 and 4, 2 and 4, 2 and 5, 3 and 5, 4 and 1), are actually adjacent in $g5$. This 5-cycle has no *chords*, and so $g5$ is not *chordal*.

Consider the 4-cycle $(1, 2, 3, 6, 1)$. There are 2 pairs of non adjacent vertices on this 4-cycle: 1 and 3, and the pair 2 and 6. The pair 1 and 3 are not adjacent in $g5$. But the pair 2 and 6 are adjacent in $g5$, and so $e = (2, 6)$ is a chord on the 4-cycle. There are many other chorded 4-cycles in $g5$, but no other unchorded n-cycles for $n \geq 4$. For example, the 5-cycle $(1, 6, 3, 4, 5, 1)$ contains the chord $(6, 4)$. If the edge $(1, 3)$ is added, there still exists the unchorded 4-cycle $(1, 3, 4, 5, 1)$ so a further edge, such as $(1, 4)$ is needed to ensure there are no unchorded n-cycles for $n \geq 4$. The graph we refer to as $g5_{dec}$ that is obtained from $g5$ by adding these 2 extra edges is depicted in Figure 3.5. Every n-cycle, $n \geq 4$, in $g5_{dec}$ has at least one chord, so $g5_{dec}$ is a chordal graph.

Making inference from the pictorial representation is easier than from the corresponding greyscale of the inverse covariance Ω. Figure 3.4 is the greyscale plot of a

generic inverse covariance Ω for $g5$. That is, $\Omega_{ij} = 0$ whenever $X_i \perp\!\!\!\perp X_j | V \backslash \{X_i, X_j\}$ for every pair of distinct vertices *not* adjacent in $g5$. Figures 3.5 and 3.6 are the pictorial and greyscale representations of the decomposable graph $g5_{dec}$, respectively. The unchorded 5-cycle is immediately obvious in Figure 3.3 but not in Figure 3.4. It is easier to assess the degree of difference between Figure 3.3 and Figure 3.5, than the degree of difference between Figure 3.4 and Figure 3.6.

Refer to the graph depicted in Figure 3.7 as $g13_{dec}$. This graph is decomposable but the graph we refer to as $g13_{nondec}$ that is obtained by adding the single extra edge $e = (a, d)$ to $g13_{dec}$ is not. The unchorded 4-cycles (a, c, i, d, a) and (a, d, h, c, a) are more obvious in Figure 3.8 than in its greyscale representation Figure 3.10.

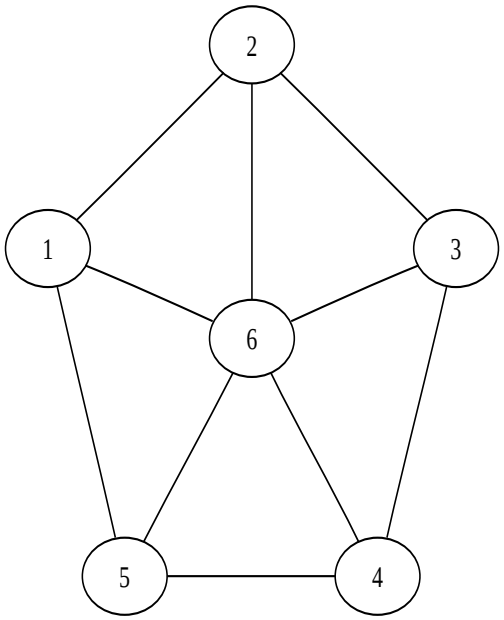


Figure 3.3: Nondecomposable graph $g5$ exhibiting a 'triangulated' appearance.

3.5.3 Finding perfect numberings, perfect sequences of cliques, and the separators, residuals and histories of the sequence

The Markov chain on the state space of all decomposable graphs with p vertices, and one step transitions given by the addition or removal of a single edge is irreducible

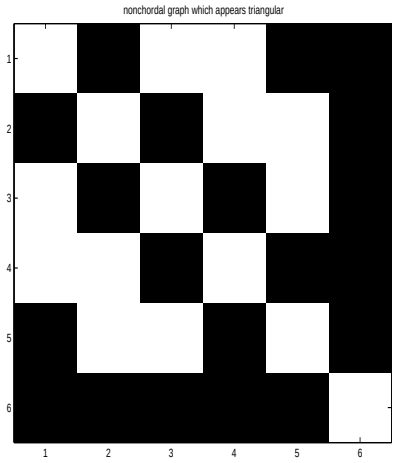


Figure 3.4: Greyscale of $g5$.

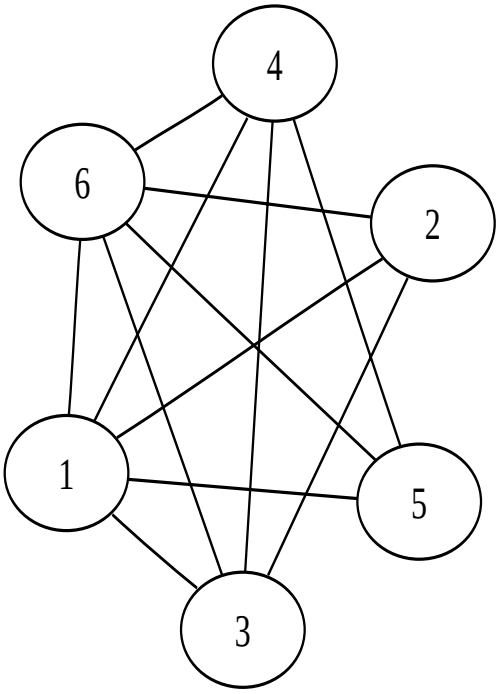


Figure 3.5: Decomposable graph $g5_{dec}$ obtained from $g5$ by adding edges (1,3) and (1,4).

(eg. Frydenberg & Lauritzen (1989)). If changing a single edge in a decomposable graph results in a decomposable graph, the edge change will be called *legal*. This section explains the graphical procedures and associated algorithms for making one step transitions via legal edge changes.

Consider $g13_{dec}$ of Figure 3.7. It contains no unchorded n-cycles for $n \geq 4$. Notice that the set of vertices $A = \{d, k, i, g, h\}$ satisfies the following 2 conditions: (1) every vertex of A is adjacent to every other vertex of A, and (2) the first condition would *not* be satisfied if a single further vertex is included in the subset. But $\{d, k, i\}$ or any subset of A only satisfies condition (1). The subset of vertices $\{a, b, c, i, h\}$ does not satisfy condition (1), even though it is the union of two sets $\{a, b, c\}$ and $\{i, h, c\}$ which each satisfy both (1) and (2). The set $\{d, j\}$ satisfies both conditions, as do $\{g, l, h, m\}, \{d, i, k, e\}$ and $\{e, f\}$. It is possible to check that these 7, and only these 7 subsets of $g13_{dec}$ satisfy both conditions. Subsets that satisfy condition (1) are said to be *complete*. Subsets which also satisfy condition (2) are said to be *maximally* complete, and are called the *cliques* of g. The set $\mathcal{C}$ of all cliques of g is uniquely defined. If g is decomposable, $\mathcal{C}$ is the collection of sets A_i which can be ordered to satisfy Sundberg's criterion, and may be arranged in a perfect sequence. The vertex indicies within each clique are the corresponding 'block' indicies in Ω. Therefore, if we can find and order the cliques and the R_i correctly, then we can find the corresponding 'block' ordering of Ω.

If the cliques can be discerned, then the process underlying the Tarjan & Yannakakis (1984) Maximum Cardinality algorithm is easy to follow and the output is a perfect numbering of the vertices $v_1, \ldots, v_p$. We now describe the algorithm. Create the variable numbering as follows. Recall that the set $nbrs(v)$ of *neighbours* of a vertex v in g is the set of all vertices adjacent to v. Choose as the next vertex v_k the vertex with the most already ordered neighbours, breaking ties arbitrarily. We use Figure 3.7 to illustrate, and refer to this graph as $g13_{dec}$. Since none of the vertices are ordered yet, choose any vertex as v_1. For example, choose $v_1 = f$. The only vertex adjacent to f is e, so set $v_2 = e$. The unordered vertices adjacent to $v_2 = e$ are $R_2 = \{d, i, k\}$ and they are all adjacent to only one already ordered vertex, e. Therefore any one of them will do for v_3. Set $v_3 = k$. Either of d or i have maximal number 2 of already ordered neighbours. So v_4 can be either d or i. Set $v_4 = i$. Maximising the number of already ordered neighbours forces the choice $v_5 = d$, the remaining

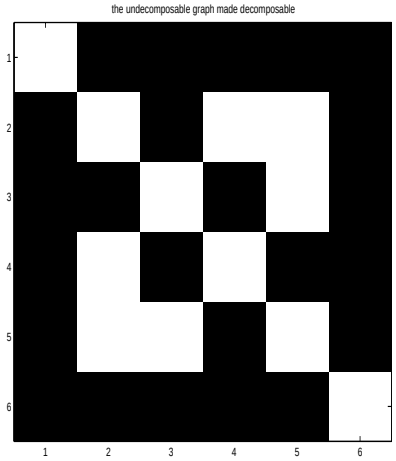


Figure 3.6: Greyscale of Figure 3.5

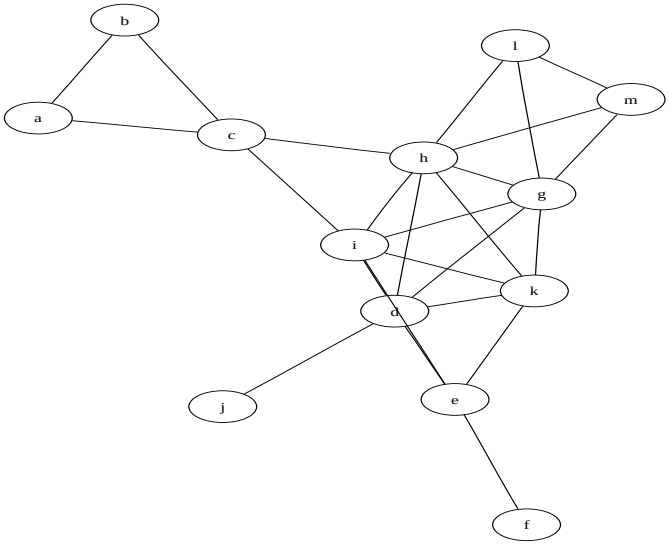


Figure 3.7: Decomposable $g13_{dec}$

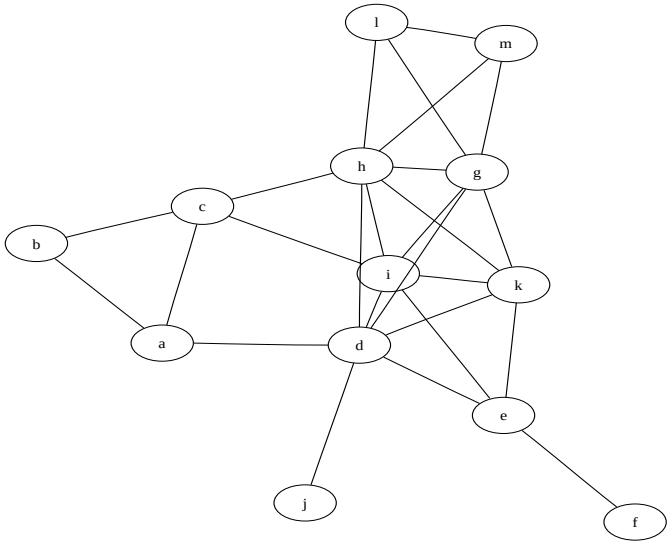


Figure 3.8: Nondecomposable $g13_{nondec} = g13_{dec} + (a, d)$

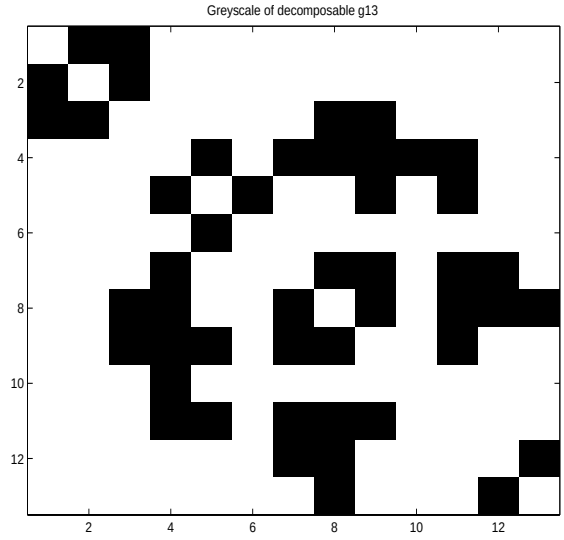


Figure 3.9: Generic Ω greyscale of decomposable $g13_{dec}$

element of the clique $C_2 = \{d, e, i, k\}$. Therefore $(v_1, \ldots, v_5) = (f, e, k, i, d)$. We must now choose $v_6 \in \{l, m, g, h, j, c, a, b\}$. None of the vertices in $\{l, m, a, b\}$ have any already ordered neighbours. Both c and j have only one already ordered neighbour, d and i respectively. Both elements of $R_3 = \{g, h\}$ have 3 already ordered neighbours $S_3 = \{k, i, d\}$. We must choose v_6 with maximal number of already ordered neighbours, so it can be either of g or h. Say $v_6 = g$. Then $v_7 = h$ is the only possibility because it has the unique maximum number of already ordered neigbours, and consequently that $(v_1, \ldots, v_7) = (f, e, k, i, d, g, h)$. Note that $v_7 = h$ was the only unnumbered element remaining from the clique $C_3 = \{k, i, d, g, h\}$. We are left to choose $v_8 \in \{c, l, m\}$. Set $v_8 = l$ so that $v_9 = m$ and $v_{10} = c$ is the only possibility which satisfies the algorithms maximal conditions. All remaining vertices $\{a, b, j\}$ have only one already ordered neighbour. If either of a or b are next numbered, we are forced to then choose the other. If j is next numbered, we can then choose either of a or b as the second to last variable in the ordering. We choose $(v_{11}, v_{12}, v_{13}) = (j, b, a)$, and the perfect numbering given by the order of the vertices in the sequence $o1 = (f, e, k, i, d, g, h, l, m, c, j, b, a)$ results. Figure 3.11 is $g13_{dec}$ with the vertices enumerated as per the numbering of vertices in the sequence $o1$. Notice that the algorithm forces you to number every vertex comprising a clique before numbering a vertex in the next clique. So if you can discern the cliques readily, the process is clearly very simple, and naturally orders the cliques. Always order the vertices in the current clique C_i before moving onto the next clique C_j. Choose the next clique as one which has largest intersection $S_j = H_i \cap C_j$ with the already ordered cliques $H_i = \cup_{k=1}^{i} C_k$, breaking ties arbitrarily. The order in which each clique is finished is the perfect sequence of Sundberg's criterion. In the previous example, the clique which had all vertices numbered first was $C_1 = \{f, e\}$. The second completed clique was $C_2 = \{e, k, i, d\}$, which is a disjoint union of 'new' variables $R_2 = \{k, i, d\}$ and 'old' variable $S_2 = \{e\}$. $H_2 = C_1 \cup C_2$ is the variables numbered so far. The third completed clique was $C_3 = \{i, d, k, g, h\}$ with $R_3 = \{g, h\}$, $S_3 = \{i, d, k\}$ and $H_3 = C_1 \cup C_2 \cup C_3$. The fourth completed clique was was $C_4 = \{g, h, l, m\}$ with $R_4 = \{l, m\}$, $S_4 = \{g, h\}$ and $H_4 = C_1 \cup C_2 \cup C_3 \cup C_4$. The fifth was $C_5 = \{i, h, c\}$ with $R_5 = \{c\}$, $S_5 = \{i, h\}$ and $H_4 = C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5$. The final two cliques were $C_6 = \{d, j\}$ and $C_7 = \{c, b, a\}$ respectively, with $R_6 = \{j\}$, $S_6 = \{d\}$, $R_7 = \{a, b\}$, $S_7 = \{c\}$. Notice that $V = C_1 \cup R_2 \cup \ldots \cup R_7$. Also notice that every numbering of

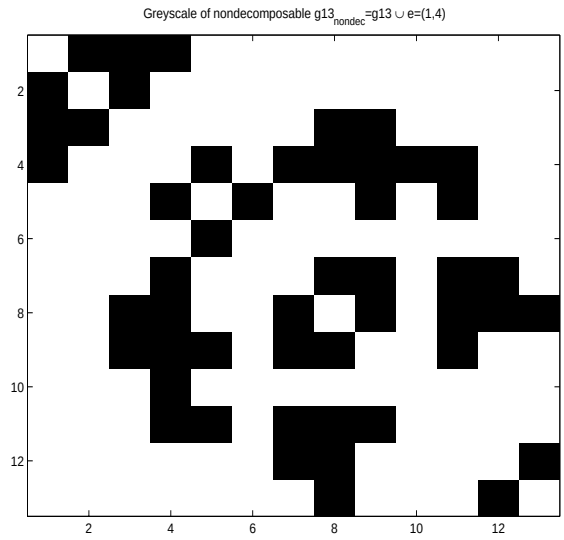


Figure 3.10: Generic Ω greyscale of nondecomposable $g13_{nondec} = g13_{dec} + (a, d)$

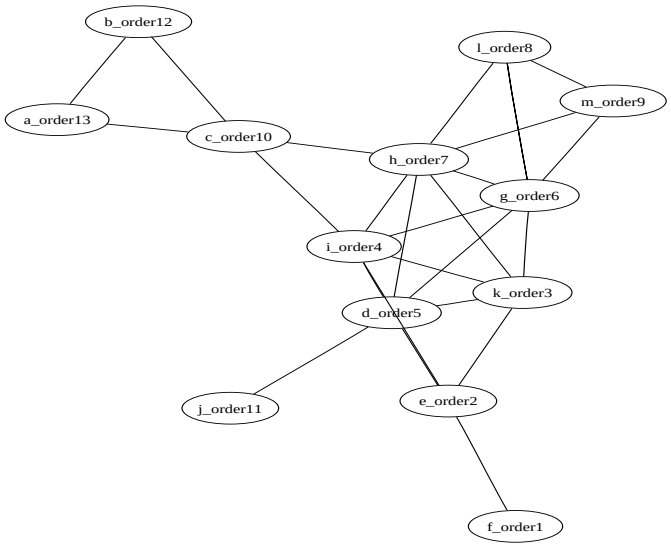


Figure 3.11: Decomposable $g13_{dec}$

the vertices in which every vertex in R_i precedes every vertex in R_{i+1} is perfect. That is, the order of the elements of each R_i is irrelevant. This example shows that the order of the R_i is not always uniquely defined, given the previous order in numbering vertices. For example, having started with $C_1 = \{f, e\}$, the set R_2 had the uniquely maximum number of already numbered neighbours, so it is necessary to next number all its members. Similarly, we are next forced to number all the elements of R_3 because these, and only these, each had maximum number 3 of previously numbered neighbours. Once all the elements of R_3, and consequently C_3, are numbered, the algorithm does not allow the elements of $R_6 = \{j\}$ to be numbered before numbering the elements of R_4 because j has only one previously numbered neighbour, and R_4 has nonunique maximal number 2 of previously numbered neighbours, $\{i, h\}$. Nor can we choose $R_6 = \{j\}$ before numbering the elements of R_5, because R_5 also has 2 previously numbered neighbours. However $R_7 = \{b, a\}$ can precede rather than follow $R_6 = \{j\}$. Each element of R_7 is adjacent to $S = \{c\}$ of size 1. Each element of R_6 is adjacent to the set of already numbered vertices $S = \{d\}$ with size 1. The key feature in this case is that R_7 and R_6 have the same maximal number of previously numbered neighbours. This illustrates that uniqueness of ordering in the sequence $\{R_i, R_{i+1}, \ldots, \}$ occurs when there is a unique maximum of already numbered neighbours. Otherwise, if there are more than one set R_i with the same maximal number of previously numbered neighbours, elements of any of these R_i can be chosen next. But having chosen this R_i, the algorithm forces you to finish numbering all its elements before beginning to number vertices in any other R_j.

We now show that this perfect sequence satisifies the running intersection property. For $j = 2$, we have $S_2 = C_2 \cap C_1 = \{e\} \subset C_1$ and $1 < j = 2$. For $j = 3$, we have $S_3 = C_3 \cap (C_1 \cup C_2) = \{k, i, d\} \subset C_2$ and $2 < j = 3$. For $j = 4$, we have $S_4 = C_4 \cap (C_1 \cup C_2 \cup C_3) = \{g, h\} \subset C_3$ and $3 < j = 4$. We leave $j = 5, \ldots, 7$ to the reader.

Figure 3.11 has vertices labelled by following the above procedure. Check that you can find alternative perfect numberings of vertices that result in different perfect sequences of the cliques, but that the sets of cliques and separators are the same in every case.

We now explain the equivalent algorithmic check of decomposability given by Tarjan & Yannakakis (1984) which is used in the code of Appendix 8.1.1. Recall

that $[v_k] = \{v_k\} \cup nbrs(v_k)$ is defined as the *closure* of v_k in g. Note that $[v_k]$ may be the same for a number of different k if $v_k \in S_i$ for at least one i. In the pictorial representation of a chordal graph g, it is obvious that if v_k is contained in only one clique (so not in any of the S_i), $[v_k]$ is that single clique. Otherwise $[v_k]$ is the union of cliques which contain v_k.

In a decomposable graph, the separators S_i are complete. The requirement that every n-cycle, $n \geq 4$ has at least one chord therefore results in the following check of decomposability. If the numbering $v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_p$ of V is perfect, then the sets $B_k = [v_k] \cap \{v_1, v_2, \ldots, v_k\}$ are always complete. This is because $[v_k]$ is either the current clique C_j or the union of cliques, some of which may come after C_j but at least one of which must precede or be equal to C_j, and the set $\{v_1, v_2, \ldots, v_k\} \subseteq H_j$. If v_k is an element of $S_i = C_k \cap C_i$ for some $i < k$ then the intersection B_k is a subset of S_i so must be complete by completeness of separators. The subset is proper unless v_k is later in the perfect numbering than the last ordered element in S_i, in which case the intersection is the separator S_i, which is complete. If v_k is any element in R_j, then B_k will be a strict subset of C_j and equal to C_j if v_k is the last element of C_j to be ordered. The completeness of cliques again gives completeness of B_k.

Compare this with the nondecomposable case. Figure 3.12 is the nondecomposable 4-cycle. No matter how the vertices are ordered, the closure of any vertex does not have complete intersection with all preceding vertices in the order. For example, if a is ordered first, then either of b or d have maximal number 1 of ordered neighbours. Choose $v_2 = b$. Then $B_2 = [b] \cap \{a, b\} = \{a, b, c\} \cap \{a, b\} = \{a, b\}$ which is complete. Either of c or d have maximal number 1 of ordered neighbours. Choose $v_3 = c$. Then $B_3 = [c] \cap \{a, b, c\} = \{b, c, d\} \cap \{b, c\} = \{b, c\}$ which is also complete. But $B_4 = [d] \cap \{a, b, c, d\} = \{c, d, a\} \cap \{a, b, c, d\} = \{c, d, a\}$ which is not complete. The addition of the chord (a, c) makes B_4 complete, and the graph chordal. Further, no ordering results in all the B_k being complete. This checking for completeness of B_k at each incremental k is the basis of the algorithm of Tarjan & Yannakakis (1984) known as maximum cardinality search (see Subsection 8.1.1).

3.5.4 Checking legality of edge changes

We first illustrate legal edge removals. Consider the graph depicted in Figure 3.13. It is obvious from the figure that the cliques are given by $C_i = \{i, i + 1, 8\}$ for

$i = 1, \ldots, 7$ and the separators by $S_i = \{i, 8\}$ for $i = 2, \ldots 6$. If any one of the edges $e_i = (i, 8), i = 2, \ldots 6$ is deleted, the unchorded 4-cycle $(i, i+1, i+2, 8, i)$ is created. Conversely, if any of the edges $(j, j+1), j = 1, \ldots, 7$ or the edge $(1, 8)$ is deleted, no unchorded n-cycle is created. Now consider the more complicated $g13_{dec}$ as given in Figure 3.7. If the edge $(g, h) \in S_4$ is deleted, then for any $x \in R_4$ and $y \in R_2$ the 4-cycle (g, x, h, y, g) is created. If the edge $(i, h) \in S_5$ is deleted, then for any $x \in R_5$ and $y \in C_3 \backslash S_5$ the 4-cycles (g, x, h, y, g) are created. As a final example, if the edge (a, b) is deleted in Figure 3.14, the unchorded 5-cycle $(a, 1, b, 2, 3, a)$ results, but any of the edges (a, i) or (b, i) for $i = 1, 2, 3$ can be removed legally.

In general, if any edge (s_1, s_2) obtained by choosing each s_1, s_2 from a separator $S_i = C_i \cap C_j$ is deleted, then a 4-cycle $(s_1, r_i, s_2, r_j, s_1)$ results, where $r_i \in C_i \backslash S_i$ and $r_j \in C_j \backslash S_j$.

We now illustrate legal edge additions. Consider $g8$ of Figure 3.13. Adding any edge $(i, i+2), i = 1, \ldots, 5$ makes this edge another chord to the 4-cycle $(i, i+1, i+2, 8, i)$ and is therefore legal. Note that each of these legal edge additions joins an element of a clique C_i and an element of a clique C_j where C_i, C_j intersect in a separator $S \in \mathcal{S}$. Furthermore, in the pictorial representation, cliques which intersect in a separator appear to be 'adjacent', or next to each other in a maximal way with respect to intersection. For example, in $g8$ of Figure 3.13 any pair of cliques have a nonempty intersection which is a subset of a separator $S \in \mathcal{S}$, but only cliques $C_i = \{8, i, i+1\}$ and $C_{i+1} = \{8, i+1, i+2\}, i = 1, \ldots 5$ look like they are 'next to each other' in a maximal way with respect to intersection. This intuitively illustrated notion of maximal intersection between cliques is equivalent to the condition that two cliques intersect in a separator, and not just the subset of a separator. The distinction between cliques having an intersection which is equal to some $S \in \mathcal{S}$ as opposed to a subset of some $S \in \mathcal{S}$ is critical in determining legal edge additions. The set of separators $\mathcal{S}$ is not defined by pairwise intersections $C_i \cap C_{i+1}$ for a given perfect sequence, but by the cumulative 'running' intersection of a clique C_j with the union of all previous cliques in the perfect sequence. However, for every $S \in \mathcal{S}$ there exists a perfect sequence of cliques such that $S = C_i \cap C_j$ for some $j < i$. In Figure 3.13, $C_j = C_{i-1}$. It is these cliques that are intuitively 'adjacent' in the graph, and that satisfy $C_i \cap C_j, j < i$ being the largest such intersection set from all C_j preceding C_i in a perfect sequence of cliques. We therefore define adjacent cliques

as follows. Let $C_1, \ldots, C_k$ be any perfect sequence of cliques. For any $i > 1$, we say that $C_j, j < i$, is an *adjacent predecessor* of C_i if $|C_i \cap C_j| \geq C_i \cap C_l$ for all $l < i$. In words, C_j is the possibly nonunique clique that precedes C_i in a perfect sequence of cliques, and has maximal sized intersection with C_i. The notion of adjacent cliques is important in graphical models. In Section 3.4 we defined $\mathcal{T}$, the junction tree of a decomposable g. $\mathcal{T}$ is a graph which has the cliques of g as its vertices, and satisfies the property that the intersection $C_i \cap C_j$ of any pair of clique vertices C_i and C_j in $\mathcal{T}$ is a subset of every clique C_l on the necessarily unique path between C_i and C_j in $\mathcal{T}$. Section 3.4 shows that $\mathcal{T}$ can be constructed sequentially from a perfect sequence $C_1, \ldots, C_k$ of cliques of g, by adding an edge between the next C_i and any of the $C_j, j < i$ such that $|C_j \cap C_i|$ is maximal with respect to all cliques that precede C_i in the sequence.

Figure 3.13 makes the distinction between '$C_i \cap C_j$ a separator' and 'clique intersection $C_i \cap C_j$ subset of separator' clear. Vertex 8 is contained in every clique, and hence every separator. Yet $\{8\} \neq S_i$ for any i. The cliques $C_1 = \{8, 1, 2\}$ and $C_2 = \{8, 2, 3\}$ are intuitively adjacent, and their intersection is a separator $S_2 = \{8, 2\}$. Intuitively, C_i is not adjacent to C_1 for any $i > 2$. Similarly, each pair of cliques $C_i = \{8, i, i + 1\}$ and $C_{i+1} = \{8, i + 1, i + 2\}$ are adjacent, but C_i is not adjacent to any $C_{i+k}, k > 1$.

Note that the sequence of separators can include repetition so more than 2 cliques can be adjacent. Figure 3.14 is a decomposable graph of 3 cliques $C_i = \{a, b, i\}, i = 1, 2, 3$ with all separators equal, and given by $S_k = \{a, b\}, k = 2, 3$.

Consider adding the edge $(7, 1)$ to the graph of Figure 3.13. This creates the unchorded 7-cycle $(1, 2, 3, 4, 5, 6, 7, 1)$ and is therefore illegal. Moreover, there are no adjacent cliques containing each of 1 and 7 respectively. Equivalently, there are no pair of cliques C_a and C_b containing $a = 1$ and $b = 7$ respectively, such that $C_a \cap C_b$ separates the vertices 1 and 7. Similarly, adding any edge $(i, i + 3), i = 1, \ldots, 4$, $(i, i + 4), i = 1, \ldots, 3$, or $(i, i + 5), i = 1, \ldots, 2$ will create an unchorded 4, 5, or 6-cycle, respectively. Again, there are no adjacent cliques containing the respective end vertices for each of these edges. Equivalently, there are no pair of cliques that contain the respective end vertices for each of these edges and whose intersection separates those same edge end vertices.

Consider $g13_{dec}$ of Figure 3.7 and the perfect sequence $C_1, \ldots C_7$ already defined.

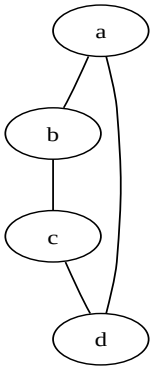


Figure 3.12: 4-cycle illustrating that B_k is not complete

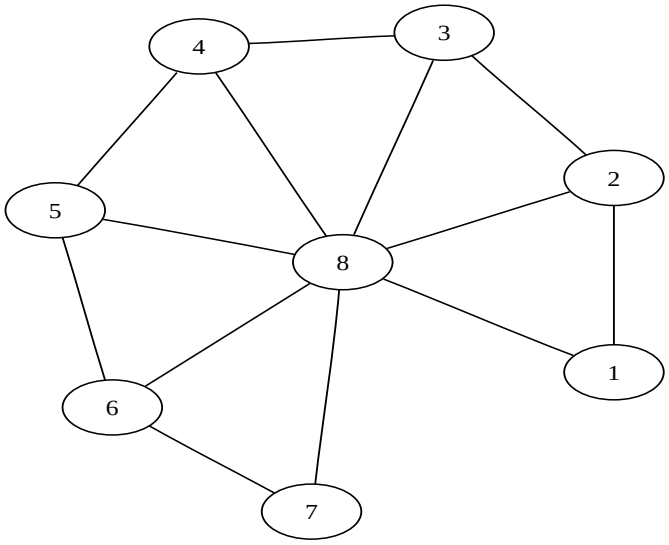


Figure 3.13: Decomposable $g8$ illustrating adjacent cliques.

Choose any 2 vertices which are not adjacent in the graph. For example, choose $g \in C_3$ and $f \in C_1$. Adding the edge (f, g) creates a number of unchorded n-cycles. For example the 4-cycle (f, e, k, g, f). However the edge addition (e, g) is legal, and $e \in C_2$ which is adjacent to C_3. The nondecomposable graph of Figure 3.8 was created from $g13_{dec}$ by adding the edge (a, d). There is no pair of adjacent cliques containing a and d respectively in $g13_{dec}$, and a number of unchorded cycles in Figure 3.8 result from adding the edge (a, d) to $g13_{dec}$.

The pictorial representation makes it obvious which cliques are adjacent. This is not the case for the greyscale or adjacency matrix representations. Furthermore, in any computational algorithms it is necessary to search amongst all pairs of cliques containing a and b respectively, because a given vertex can be in a number of cliques. But it is immediately obvious in a diagram whether two cliques are adjacent or not. For example, e is also in C_1. But C_1 is clearly not adjacent to C_3. We have just shown that the edge (e, g) is a legal addition and that e and g are also contained in adjacent cliques C_2 and C_3 respectively.

Consider again Figure 3.13. Recall that adding any edge $(i, i+2), i = 1, \ldots, 5$ is legal. A perfect sequence is given by $\{C_i = \{i, i+1, 8\} : i = 1, \ldots 6\}$. Consider $i = 1$ and the edge addition $(1, 3)$. Every path connecting 1 and 3 includes at least one element of $C_1 \cap C_2 = \{2, 8\}$. Intuitively from the diagram, the elements in $C_1 \cap C_2$ are 'gates' on every path between 1 and 3. If these 'gates' were 'shut', then 1 would be 'separated' from 3. Intuitively, every path between 1 and 3 would be blocked. Similarly, for every legal edge addition $(i, i+1)$, every path between i and $i+1$ must include an element of $C_i \cap C_{i+1} = \{i, 8\}$. Intuitively from the diagram, the set $\{i, 8\}$ can be used to block every path connecting i and $i+1$. Now consider an illegal edge addition such as between 1 and 4. Then $C_1 \cap C_j = \{8\}$ for all the $C_j, j = 3, 4$ which contain 4. But there exists a path $1 - 2 - 3 - 4$ connecting 1 and 4 which does not include an element of $C_1 \cap C_j = \{8\}$. Intuitively, neither of the cliques C_j satisfy the requirement that $C_1 \cap C_j$ blocks all the paths between 1 and 4. It is this notion of 'path blocking' that is formalised by the definition of (u, v)-separation.

The essential feature and advantage of the pictorial representation is as an inference tool for checking independence and conditional independence in a distribution it is assumed to represent. It is the relative ease with which these can be discerned from the diagram as opposed to the distribution (or the covariance in distributions

where these are equivalent) that makes the pictorial representation so attractive. We therefore give more examples of graph separation and its use in checking legality of edge additions for decomposable graphs.

Consider $g13_{dec}$ of Figure 3.7 and the illegal edge addition between f and g. Now $f \in C_1$ only, while g is contained in 2 cliques, C_3 and C_4. Both intersections $C_i \cap C_j$ for $j = 3, 4$ are empty. $D = \emptyset$ does not separate any vertices, so the edge is illegal as already shown. Adding an edge between a and d is similarly shown to be illegal. Here $a \in C_7$ only, while $d \in C_j$ for $j = 2, 3, 6$, and each of $C_7 \cap C_j = \emptyset$ for $j = 2, 3, 6$. If an edge is added between m and c, a 4-cycle (m, c, i, g, m) is created. Check that none of the pairs m and i, c and g, or i and m which are not consecutive on this cycle, are adjacent in $g13_{dec}$. Hence (m, c, i, g, m) is an unchorded 4-cycle showing that the edge addition results in a nondecomposable graph. Lemma 3.3.4 gives the same result, as $m \in C_4$ only and the only clique containing a which has nonempty intersection with C_4 is $C_5 = \{h, i, c\}$. Here $C_4 \cap C_5 = \{h\}$, but there are many paths connecting m and c which do not include h. One example is the path (m, g, i, c). Therefore the conditions of Lemma 3.3.4 fail as required.

Now consider adding an edge between m and k. No unchorded n-cycles for $n \geq 4$ result: i.e. this is a legal edge addition. In this case, $C_4 \cap C_3 = \{g, h\}$, and there is no path between m and c which does not include at least one member of $C_4 \cap C_3$. Lemma 3.3.4 implies this is a legal edge addition as required.

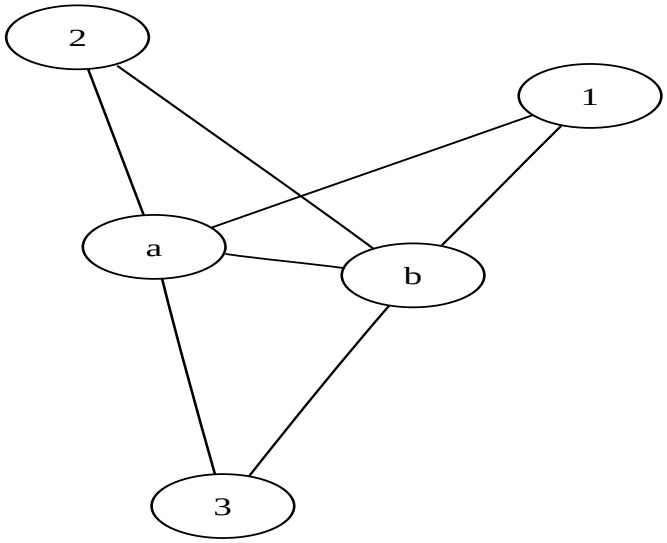


Figure 3.14: Decomposable $g5$ illustrating all cliques adjacent, so all separators equal and given by $S_k = \{a, b\}, k = 1, 2$.

Chapter 4

Bayesian covariance selection models

4.1 Likelihood and hierarchical structure

Suppose we have independent observations

$$y_t \sim N(\mu, \Sigma), \quad t = 1, \ldots, n, \tag{4.1}$$

where y_t is $p \times 1$. Let $y = (y_1, \ldots, y_n)$ be the data. We use a hierarchical prior for μ and Σ of the form

$$p(\mu, \Sigma, \Phi, \delta, g) = p(\mu|\Sigma, \Phi, \delta, g)p(\Sigma|\Phi, \delta, g)p(\Phi|\delta, g)p(\delta|g)p(g),$$

where each of the terms on the right is discussed below. We assume that $p(\mu|\Sigma, \Phi, \delta, g) \propto c$ for some constant c, as our focus in this chapter is on priors for Σ. The prior for Σ depends on its graph g, the $p \times p$ matrix Φ and the scalar δ, and is discussed in Section 4.5. Section 4.2 defines the graph of Σ as the configuration of nonzero off-diagonal elements in Σ^{-1}. The prior for Φ is discussed in Section 4.6 and the prior for the graph g is discussed in Section 4.7. We set the degrees of freedom parameter δ to 5 as such a value of δ gives a suitably noninformative prior for Σ.

We restrict the graph of Σ to be decomposable, so that the prior for Σ is a mixture over all decomposable graphs. We explain in Section 4.2 that this is equivalent to the prior for $\Omega = \Sigma^{-1}$ being a mixture over all Wishart distributions constrained to decomposable graphs.

4.2 The hyper inverse Wishart distribution.

Let $g = (V, E)$ be an undirected graph with vertices $V = \{1, \ldots, p\}$ and set of edges $E \subseteq V \times V$, where an element of E is given by the index pair (i, j) of the vertex pair $(V_i, V_j) \in V \times V$. For a square matrix A we write $A > 0$ to denote that A is positive definite. Let $M^+(g)$ be the set of $p \times p$ matrices Ω satisfying $\Omega > 0$ and $\Omega_{ij} = 0$ for all pairs $(i, j) \notin E$.

For a given $p \times p$ covariance matrix Σ, we define the *graph* $g = g(\Sigma)$ of Σ, as follows. Let $\Omega = \Sigma^{-1}$. Let $V = \{1, \ldots, p\}$ and define $E = \{(i, j), i \neq j$ such that $\Omega_{ij} \neq 0\}$. Thus the graph $g(\Sigma)$ gives the configuration of nonzero off-diagonal elements in Ω.

We say that an $m \times m$ matrix $A > 0$ has an inverse Wishart (IW) density with $\delta > 0$ degrees of freedom and scale matrix Φ, denoted as $A \sim IW(m, \delta, \Phi)$, if the density of A is

$$p(A|\delta, \Phi) = \frac{|\Phi|^{\frac{\delta}{2}}}{2^{\frac{m\delta}{2}}\Gamma_m(\frac{\delta}{2})}|A|^{-\frac{(\delta+m+1)}{2}} \operatorname{etr}\left(-\frac{1}{2}\Phi A^{-1}\right) \tag{4.2}$$

where $\operatorname{etr}(A) = \exp(\operatorname{trace}(A))$ and for $\alpha > \frac{(m-1)}{2}$,

$$\Gamma_m(\alpha) = \pi^{m(m-1)/4} \prod_{i=1}^{m} \Gamma(\alpha - (\frac{i-1}{2}))$$

is the multivariate gamma function (Muirhead, 1982,p. 113). We will refer to $g = g(\Sigma)$, and say that Σ and Ω are decomposable if $g = (V, E)$ is decomposable.

Suppose that g is a decomposable graph and let $C_1, \ldots, C_k$ be a perfect sequence of the cliques of g. Let $H_j = C_1 \cup \ldots \cup C_j$ be the history of the sequence and let $S_j = H_{j-1} \cap C_j$ be the separators for $j = 2, \ldots, k$. For any matrix M and subset of vertices B, use M_{BB} to denote the symmetric submatrix of M which is formed by taking every corresponding entry M_{ij} for which the vertices $\{V_i, V_j\} \in B$. Using the parameterization of Dawid (1981), we say Σ has a *hyper inverse Wishart* (HIW) distribution, with *hyperparameters* (δ, Φ) denoted by $\Sigma \sim HIW(g, \delta, \Phi)$, if for $\Sigma^{-1} \in M^+(g)$

$$p(\Sigma|\delta, \Phi, g) = \frac{\prod_{i=1}^{k} p\left(\Sigma_{C_iC_i}|\delta, \Phi_{C_iC_i}\right)}{\prod_{i=2}^{k} p\left(\Sigma_{S_iS_i}|\delta, \Phi_{S_iS_i}\right)} \tag{4.3}$$

where $\delta > 0$, $\Phi > 0$, and the density is with respect Lebesgue measure on the elements of Σ corresponding to edges of g.

In (4.3), the terms $p\left(\Sigma_{C_iC_i}|\delta, \Phi_{C_iC_i}\right)$ denote the IW densities $\Sigma_{C_iC_i} \sim IW\left(|C_i|, \delta + |C_i| - 1, \Phi_{C_iC_i}\right)$ given by

$$p\left(\Sigma_{C_iC_i}|\delta, \Phi_{C_iC_i}\right) = \frac{\left|\frac{\Phi_{C_iC_i}}{2}\right|^{\left(\frac{\delta+|C_i|-1}{2}\right)}}{\Gamma_{|C_i|}\left(\frac{\delta+|C_i|-1}{2}\right)} \left|\Sigma_{C_iC_i}\right|^{-\left(\frac{\delta+2|C_i|}{2}\right)} \operatorname{etr}\left[-\frac{1}{2}\left(\Sigma_{C_iC_i}\right)^{-1}\Phi_{C_iC_i}\right] \tag{4.4}$$

where $|C_i|$ denotes the cardinality of the clique C_i, and the terms $p\left(\Sigma_{S_iS_i}|\delta, \Phi_{S_iS_i}\right)$ are defined similarly. Note that the expression in (4.3) is invariant to the choice of perfect sequence.

From (4.2) – (4.4), the normalizing constant for the HIW distribution is

$$h(g, \delta, \Phi) = \frac{\prod_{i=1}^{k}\left[\left|\frac{\Phi_{C_iC_i}}{2}\right|^{\left(\frac{\delta+|C_i|-1}{2}\right)}\Gamma_{|C_i|}\left(\frac{\delta+|C_i|-1}{2}\right)^{-1}\right]}{\prod_{i=2}^{k}\left[\left|\frac{\Phi_{S_iS_i}}{2}\right|^{\left(\frac{\delta+|S_i|-1}{2}\right)}\Gamma_{|S_i|}\left(\frac{\delta+|S_i|-1}{2}\right)^{-1}\right]}. \tag{4.5}$$

4.3 Generating a Markov chain from the HIW distribution

This section shows how to generate the covariance matrix Σ of a decomposable Gaussian distribution efficiently from its prior or posterior by conditioning on a decomposable graph g. The computer programs in Sections 8.1.15 and 8.1.16 sample $\Sigma \sim HIW(g, \delta, \Phi)$, and are based on the theory in this section.

Section 3.3 shows how to transition from one decomposable graph to another by either adding or removing an edge. Combined with a decision rule on whether to make the transition, and given the first iterate $g^{[1]} = (V, E)$, this methodology is all that is required to generate a Markov chain of decomposable graphs $g^{[1]}, g^{[2]}, \ldots$. None of the procedures depends on a particular distribution: the only condition is the decomposability of g. Hence, given any decomposable $g = (V, E)$, graphical procedures can be used to sample from any distribution which is globally Markov over g. Whenever each graph dependent normalising constant $p_V(x_V|g)$ can be calculated,

then the variables X_V can be integrated out to give a sampler for g. In this case the variables X_V are like a hierarchical conjugate prior for g, which is the case for the $\Sigma \sim HIW(g, \delta, \Phi)$.

Let θ be the parameter of a distribution for X_V, and let $P(\theta|g)$ be the distribution of θ. Dawid & Lauritzen (1993) coined the term *laws* for distributions of parameters of random variables associated with the vertices of a graph and the adjective *hyper* for all the parameter analogous concepts. If the parameters of the marginals (when considered as random variables) satisfy the graph implied independencies so that $P(\theta|g)$ factorises according to g, then $P(\theta|g)$ is *hyper Markov* with respect to g.

Section 2.5 shows that Σ can be uniquely defined by specifying each Σ_{ij} for all $(i,j), i = j$ or $(i,j) \in E$, and requiring that $\Sigma^{-1} = \Omega \in M^+(g)$. A covariance selection model is a regular exponential family model with canonical statistic $K \in M^+(g)$ (Lauritzen, 1996,p. 132). Therefore, Σ can be sampled via the canonical statistic Ω. The Wishart distribution is the probability distribution of the maximum likelihood estimator of the covariance matrix of a multivariate normal distribution. It is a conjugate prior and so a g-constrained version of the Wishart is a natural choice for Ω, and a g-constrained version of the inverse Wishart is a natural choice for Σ. A g-constrained version of the inverse Wishart which has clique consistent marginals can be defined analogously to the way in which a measure on the line $ax + by = c$ in $\mathbb{R}^2$ can be defined by using the parameterisation $x = t$, $y = (c - at)/b$, $r(t) = (t, c - at/b) \in R^2$ and letting $P(dt) = p(t)dt$ where $p(t) = |dr/dt| = |(dx/dt, dy/dt)| = \sqrt{1 + a^2/b^2}$. To explain further, let $M^+(g)$ be the set of $p \times p$ matrices which are positive definite and have zero ijth entry for all pairs $(i,j) \notin E$. If $g = (V, E)$, then an analogous parameterisation of the manifold $M^+(g)$ is obtained by letting $t_1, \ldots, t_{|E|}$ correspond to 'the edges in Σ'; i.e. each t_k corresponds to exactly one $\Sigma_{i,j}$ such that $(i,j) \in E$. The *hyper inverse Wishart* (HIW) measure on $M^+(g)$ can then be defined by $P_{HIW}(d(t_1, \ldots, t_{|E|}) = p_{HIW}(t_1, \ldots, t_{|E|})d(t_1, \ldots, t_{|E|})$ where p_{HIW} is given in Section 4.5 and $d(t_1, \ldots, t_{|E|})$ is the usual Lesbegue measure in Euclidean space. However, such distributions have normalizing constants that are not available analytically unless the graph is decomposable.

Let θ_V be the parameters of a distribution considered as random variables, and let P_{θ_V} be the associated hyper distribution. For any subsets A, B of V, let P_{θ_A} be the marginal distribution of θ_A and let $P_{\theta_{B|A}}$ be the distribution of θ_B conditional

on θ_A. We say that P_θ is *strong hyper Markov* over $g = (V, E)$ if $\theta_{B|A} \perp\!\!\!\perp \theta_A$ for any decomposition (A, B) of g.

If g is decomposable and the marginals of P_{θ_V} are consistent, then P_θ factorises according to g and so is *strong hyper Markov*. In which case, the next three theorems can be used to sample Σ from its exact HIW distribution.

Theorem 4.3.1 *(Roverato, 2000,Theorem 2, p.105) For the parameter Σ of a decomposable covariance selection model with graph $g = (V, E)$, the following are equivalent:*

(i) the distribution of Σ is strong hyper Markov over g;

(ii) for every enumeration of the vertices V following a perfect elimination scheme for g, the rows of the matrix Ψ defined by $\Sigma^{-1} = \Psi'\Psi$ are mutually independent.

Propositions 3, 4, and 5 of Wermuth (1980) together assert that every decomposable covariance selection model with covariance Σ can, after proper reordering of the variables, be characterised by Ψ, where $\Sigma^{-1} = \Psi'\Psi$ is the Cholesky decomposition of Σ^{-1}. In particular, if the vertices V are reordered to follow any perfect elimination scheme for g, then the zero pattern of the upper triangle of Ψ is the same as the zero pattern of Σ^{-1}, and even though there is more than one perfect elimination scheme for a given g, the elements of Ψ do not depend on the choice (see Paulsen et al. (1989)).

A perfect elimination scheme for a graph g with cliques $C_1, \ldots, C_k$ and residuals $R_2, \ldots, R_k$ is given as follows. First, take the variables in C_1 in any order. Follow these by the variables in R_2 in any order. Continuing, the sequence given by $C_1, R_2, \ldots, R_k$ is a perfect numbering (Lauritzen, 1996,Lemma 2.21, p. 15). A perfect elimination scheme is given by reversing this sequence. Hence Theorem 4.3.2 given below can be used to generate $\Sigma \sim HIW(g, \delta, I)$ for I the identity matrix. Note that I corresponds to the graph of no edges in which all variables are trivially separated. Let $pa(\alpha_i)$ be the adjacent predecessors of the ith variable in the perfect order just described. This is, equivalently, the adjacent variables that follow the ith variable in the perfect elimination scheme.

Theorem 4.3.2 *(Roverato, 2000,Theorem 3) Suppose $\Sigma_{id} \sim HIW(g, \delta, I)$, where $g = (V, E)$ is decomposable and $\delta > 0$ is an integer. Let $(\Sigma_{id})^{-1} = \Psi'_{id}\Psi_{id}$ be the*

Cholesky decomposition of $(\Sigma_{id})^{-1}$*, and let* $pa(\alpha_i)$ *be the parents of the ith variable in the perfect ordering given by* $C_1, R_2, \ldots, R_k$*. Then the main diagonal elements* $(\Psi_{id})_{ii}$ *are square roots of* $\chi^2_{(\delta+|pa(\alpha_i)|)}$ *quantities, the off-diagonal elements* $(\Psi_{id})_{rs}$ *with* $r < s$ *and* $(\alpha_r, \alpha_s) \in E$*, are* $N(0,1)$*, and these random variables are mutually independent.*

In order to sample $\Sigma \sim HIW(g, \delta, \Phi)$, where $\Phi \in M^+(g)$, a closed transformation of the hyper inverse Wishart space is performed using Theorem 4.3.3.

Theorem 4.3.3 *(Roverato, 2000,Theorem 4) Let* $g = (V, E)$ *be a decomposable graph with cliques* $C_1, \ldots, C_k$*, separators* $S_2, \ldots, S_k$ *and residuals* $R_2, \ldots, R_k$*. Let* D *be a* $|V| \times |V|$ *positive definite matrix, and suppose* $\Sigma \sim HIW(g, \delta, B)$ *where* $\delta > 0$ *is an integer and* $B^{-1} \in M^+(g)$*. Assume that the vertices of* g *are enumerated according to a perfect elimination scheme, and similarly for the entries in* B *and* D*. Put* $\Sigma^{-1} = \Psi'\Psi$*, and for* $j = 1, \ldots, k$*, successively define the Cholesky decompositions* Q^j, P^j *and the matrices* O^j *by putting:*

$$(B_{C_j,C_j})^{-1} = (Q^j)'Q^j,$$

$$(D_{C_j,C_j})^{-1} = (P^j)'P^j,$$

and

$$O^j = (Q^j)^{-1}P^j.$$

Then the upper triangular matrix Υ *defined by*

$$\Upsilon_{C_1,C_1} = \Psi_{C_1,C_1}O^1,$$

and for $j = 1, \ldots, k$

$$\Upsilon_{R_j,R_j} = \Psi_{R_j,R_j}O^j_{R_j,R_j}, \Upsilon_{R_j,S_j} = \Psi_{R_j,R_j}O^j_{R_j,S_j} + \Psi_{R_j,S_j}O^j_{S_j,S_j}$$

is such that $(\Upsilon'\Upsilon)^{-1} \sim HIW(g, \delta, D)$*.*

Theorems 4.3.2 and 4.3.3 are consequences of the standard Odell & Feiveson (1966) result for Wishart distributions:

Theorem 4.3.4 *Odell & Feiveson (1966) Let* $\Omega \sim W(\delta + |V| - 1, I)$ *have Cholesky decomposition* $\Omega = \Psi'\Psi$*. Then the main diagonal elements* Ψ_{ii} *are square roots of*

$\chi^2_{(\delta+|V|-i)}$ quantities, the off-diagonal elements $\Psi_{ij} \sim N(0,1)$ and these random variables are mutually independent. A Wishart matrix with parameter $A = T'T$ may successively be obtained by the transformation $(\Psi T)'(\Psi T) = T'\Omega T \sim W(\delta + |V| - 1, A)$.

4.4 HIW results for Bayesian analysis using MCMC

We first define notation for the *edge indicators* of a graph g. Let

$$e_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

and let $e_{-ij} = \{e_{kl} : (k,l) \neq (i,j)\}$. Note that any graph $g = (V, E)$ can be unambiguously written as $g = (e_{ij}, e_{-ij})$.

For a given graph $g = g(\Sigma)$, let the number of edges, or the *size* of g, be given by

$$size(g) = \sum_{i<j} e_{ij} \tag{4.7}$$

i.e. $size(g)$ is the number of nonzero elements in the strict upper triangle of Ω, and $size(g) \leq r = p(p-1)/2$.

Section 4.9 describes a reduced conditional Metropolis Hastings sampler for g. The covariance selection code of Section 8.1.23 is based on the programs in Sections 8.1.21 and 8.1.22 and recalculates the perfect sequence of each new graph iterate in the chain so that the legality of the next proposed edge change can be checked. If it were possible to find a perfect sequence of the cliques of any graph g' from the cliques of a given graph g which only differed from g' by a single edge, then a far more computationally efficient program results.

The results in this section show that this is possible, and furthermore, that there is a consequent expression for the graph likelihood given in (4.18) that can be evaluated efficiently. The next theorem summarises the necessary graph theory for updating the perfect sequence of g to give the perfect sequence of g'.

Let $g = (V, E)$ be a decomposable graph with edge indicators $\{e_{ij}, i < j \leq p\}$. Assume the edge indicator $e_{ij} = 1$ for g, and that the graph $g' = (V, E')$ is decomposable and has edge set E' as defined by indicators $\{e'_{ij} = 0, e_{-ij}\}$. So g' is obtained from g by deleting the edge (i,j). The following theorem is used for gains in efficiency in the reduced conditional sampler.

Theorem 4.4.1 *(Wong (2002)). Suppose that g and g' are the decomposable graphs defined above. Suppose that $C_1, \ldots, C_k$ are the cliques of g ordered to form a perfect sequence and $S_2, \ldots, S_k$ are the corresponding separators. Then*
(a) The edge (i, j) is contained in a single clique of g (Giudici & Green (1999)).
(b) If $(i, j) \in C_q$ then either $i \notin S_q$ or $j \notin S_q$.
(c) If $j \notin S_q$ and $C_{q_1} = C_q \backslash \{j\}$ and $C_{q_2} = C_q \backslash \{i\}$ then C_1, ..., C_{q-1}, C_{q_1}, C_{q_2}, C_{q+1}, ..., C_k is a perfect sequence of complete sets in g' and has separators S_2, ..., S_{q-1}, $S_{q_1} = S_q$, $S_{q_2} = C_q \backslash \{i, j\}$, S_{q+1}, ..., S_k.
(d) The sequence C_1, ..., C_{q-1}, C_{q_1}, C_{q_2}, C_{q+1}, ..., C_k contains all the cliques of g'.

Remark 4.4.2 *Part (a) is Theorem 1 of Giudici & Green (1999).*
Parts (b) and (c) follow from part (a) and Lemma 2.20 of Lauritzen (1996).

Two instructive examples are given to illustrate Theorem 4.4.1 because the efficiency of the reduced conditional sampler depends on its application. Before giving the examples, the gains in efficiency are explained. The characterisation of the cliques and separators in Theorem 4.4.1 means that it is unnecessary to compute the marginal likelihoods for $g^{[i]}$ and $g^{[i+1]}$ (or their normalising constants). Section 4.4 gives an expression for the ratio $p(y|g^p, \delta, \Phi)/p(y|g^c, \delta, \Phi)$ of the posterior likelihoods in the MH acceptance probability for the transition proposal. Theorem 4.4.1 results in an expression for $p(y|g^p, \delta, \Phi)/p(y|g^c, \delta, \Phi)$ that involves the subgraphs S_{q2} and e only: this is given in (4.8) of Lemma 4.4.3. Lemma 4.4.5 gives an efficient method for calculating (4.8). The program to calculate $\frac{h(g,\delta,\Phi)}{h(g',\delta,\Phi)}$ in $p(y|g^p, \delta, \Phi)/p(y|g^c, \delta, \Phi)$ is given in Section 8.1.18.

Theorem 4.4.1 is also the basis of an efficient method for computing a perfect sequence of cliques for the one step transition state $g^{[i+1]}$, given a perfect sequence of cliques for the ith graph iterate $g^{[i]}$. This 'local update' methodology based on Theorem 4.4.1 is far less computationally expensive than the methodology used by Giudici & Castelo (2003), who use the algorithm of Tarjan & Yannakakis (1984) with complexity $\mathcal{O}(|V| + |E|)$ to update the clique sequence with each transition.

The illustrative examples are now given. Using the notation of Theorem 4.4.1, let $i = u$, $j = v$, and consider the graph $g^0 = (e_{uv}, e_{-uv})$ where $e_{-uv} = \{e_{ab} = 1, e_{au} = 1, e_{ar} = 1, e_{av} = 1, e_{bu} = 1, e_{br} = 1, e_{bv} = 1, e_{ur} = 1\}$. First consider the case where $e_{uv} = 0$, and $g' = (e_{uv} = 0, e_{-uv})$ as depicted by the first graph of

Figure 4.1. The dotted line represents the legal edge addition $e = (u, v)$ to g'. A perfect sequence of cliques for g' is $\{C_u = \{a, b, u, r\}, C_v = \{a, b, v\}\}$ with separator $S_v = C_v \cap (C_u \cup C_v) = \{a, b\}$. If the edge $e = (u, v)$ is added, then the resulting $g = (e_{uv} = 1, e_{-uv})$ has new clique $C_q = (C_u \cap C_v) \cup e = \{a, b, u, v\} = S_v \cup e$, and a new separator $S_q = \{a, b, u\}$. A perfect sequence for $g = (e_{uv} = 1, e_{-uv})$ is $\{C_u, C_q\}$. This illustrates that the clique C_q containing the extra edge e and required to define S_{q2} in Lemmas 4.4.3 and 4.4.5 is given by $(C_u \cap C_v) \cup e$, where C_u, C_v are the cliques of Lemma 3.3.4. The output clique C of the MATLAB routine in Sections 8.1.12 is computed using this fact. The corresponding qth separator is given by $S_q = C_u \cap C_v$, where C_u, C_v are the cliques of Lemma 3.3.4. Note that S_q is the separator S^* constructed in the proof of Lemma 3.3.4.

Now consider the case where the edge $e = (u, v)$ is deleted from $g = (e_{uv} = 1, e_{-uv})$. Refer to the second graph in Figure 4.1, in which the dotted line indicates the edge (u, v) to be deleted. The edge $e = (u, v)$ is contained in a single clique $Cq = \{a, b, u, v\}$ of g as required by Theorem 4.4.1(a). The output clique C of the MATLAB routine given in Section 8.1.11 is this clique C_q. In accordance with Theorem 4.4.1(b), $e \in C_q$, and $i = u \in S_q = \{a, b, u\}$, but $j = v \notin S_q$. In accordance with Theorem 4.4.1(c), $C_{q1} = C_q \backslash \{j\} = C_q \backslash \{v\} = \{a, b, u, v\} \backslash \{v\} = \{a, b, u\}$, and $C_{q2} = C_q \backslash \{i\} = C_q \backslash \{u\} = \{a, b, u, v\} \backslash \{u\} = \{a, b, v\} = C_v$. The sequence $\{C_u = \{a, b, u, r\}, C_{q1} = \{a, b, u\}, C_{q2} = C_v = \{a, b, v\}\}$ is a perfect sequence of complete sets in $g' = (e_{uv} = 0, e_{-uv})$. Note that the sequence is not a sequence of cliques, as the complete set C_{q1} is not maximal (it is contained in $C_u = \{a, b, u, r\}$). The corresponding sequence of separators for the sequence of complete sets is $\{S_{q1} = S_q = \{a, b, u\}, S_{q2} = C_q \backslash \{u, v\} = \{a, b\}\}$. In accordance with Theorem 4.4.1(d), the sequence $\{C_u, C_{q1}, C_{q2} = C_v\}$ contains the perfect sequence $\{C_u, C_v\}$ of cliques of $g' = (e_{uv} = 0, e_{-uv})$. Lemma 4.4.4 can then be applied to the sequence $\{C_u, C_{q1}, C_{q2} = C_v\}$ to obtain the perfect clique sequence $\{C_u, C_v\}$.

In the preceding example of Figure 4.1, the clique C_q that contains the extra edge $e = (u, v)$ subsumes the clique C_v so $C_{q2} = C_v$. This is because the residual R_v consists solely of v, one of the edge ends of the extra edge $e = (u, v)$. This is not the case whenever there exists at least one extra vertex k in C_v, as illustrated by Figure 4.2. Here $e_{-uv} = \{e_{ab} = 1, e_{au} = 1, e_{ar} = 1, e_{av} = 1, e_{ak} = 1, e_{bu} = 1, e_{br} = 1, e_{bv} = 1, e_{bk} = 1, e_{ur} = 1, e_{v,k} = 1\}$, and $g = (e_{uv} = 1, e_{-uv})$ has perfect sequence

of cliques $\{C_u, C_q, C_v\}$ where the cliques $C_u = \{a, b, u, r\}$ and $C_q = \{a, b, u, v\}$ are as before, but $C_v = \{a, b, v, k\}$ is no longer subsumed by C_q. Parts (a) and (b) of Theorem 4.4.1 are verified because $e \in C_q$ only, and $j = v \notin S_q = \{a, b, u\}$. The sets $C_{q1} = C_q \backslash \{v\} = \{a, b, u\}$ and $C_{q2} = C_q \backslash \{u\} = \{a, b, v\}$ are the same as in the previous example of Figure 4.1, but now $C_{q2} = \{a, b, v\}$ is a proper subset of $C_v = \{a, b, v, k\}$. Hence $C_{q2} \neq C_v$ in the sequence $\{C_u, C_{q1}, C_{q2}, C_v\}$. In accordance with Theorem 4.4.1(c) this sequence is a perfect sequence of complete sets in $g' = (e_{uv} = 0, e_{-uv})$ with separators $S_{q1} = S_q = \{a, b, u\}, S_{q2} = C_q \backslash \{u, v\} = \{a, b\}, S_v = \{a, b\}$. Finally, the perfect sequence $\{C_u, C_v\}$ of cliques of g' is contained in the sequence $\{C_u, C_{q1}, C_{q2}, C_v\}$ as required by Theorem 4.4.1(d), and can be obtained from the sequence $\{C_u, C_{q1}, C_{q2}, C_v\}$ by applying Lemma 4.4.4.

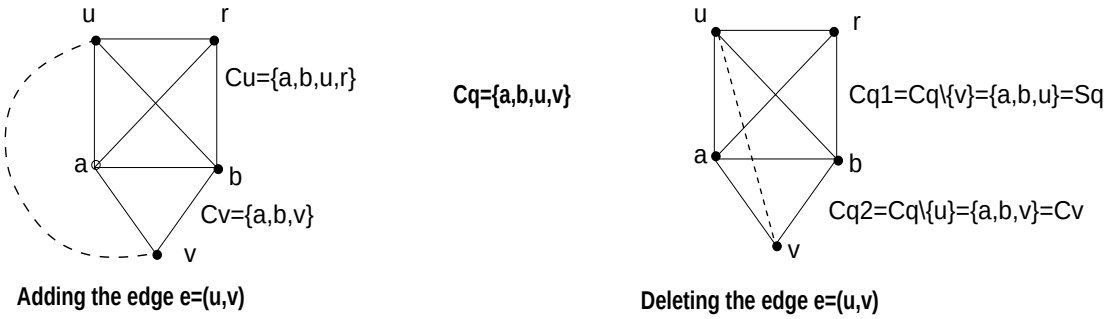


Figure 4.1: Illustration of Theorem 4.4.1 for the case in which the residual R_v consists of just the edge end vertex v.

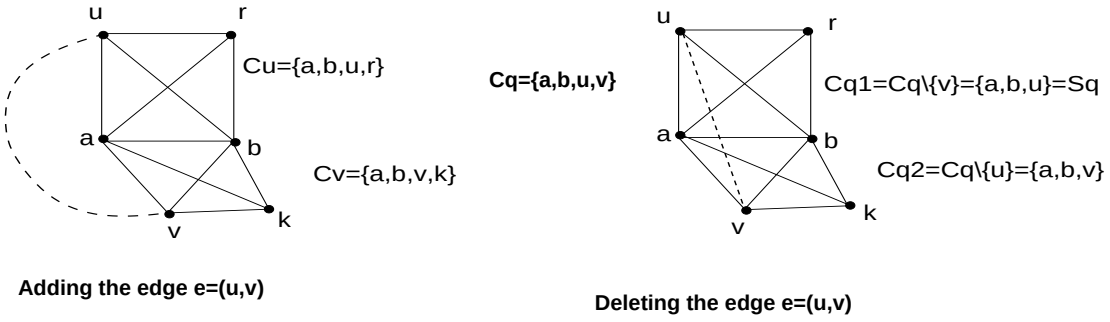


Figure 4.2: Illustration of Theorem 4.4.1 for the case in which the residual R_v consists of at least 1 additional vertex k to the edge end vertex v.

The next lemma uses (4.5) and Theorem 4.4.1 to simplify the expression for the likelihood ratio in (4.18).

Lemma 4.4.3 *(Wong (2002)). Suppose that g and g' are the decomposable graphs defined above. Then, using the notation of (4.15), and Theorem 4.4.1*

$$\begin{aligned}
&\frac{h(g,\delta,\Phi)}{h(g',\delta,\Phi)}\frac{h(g',\delta^*,\Phi^*)}{h(g,\delta^*,\Phi^*)} \\
&\quad=\frac{\left|\Phi_{DD|S_{q_2}}\right|^{\left(\frac{\delta+|S_{q_2}|+1}{2}\right)}\left|\Phi^*_{ii|S_{q_2}}\right|^{\left(\frac{\delta^*+|S_{q_2}|}{2}\right)}\left|\Phi^*_{jj|S_{q_2}}\right|^{\left(\frac{\delta^*+|S_{q_2}|}{2}\right)}}{\left|\Phi_{ii|S_{q_2}}\right|^{\left(\frac{\delta+|S_{q_2}|}{2}\right)}\left|\Phi_{jj|S_{q_2}}\right|^{\left(\frac{\delta+|S_{q_2}|}{2}\right)}\left|\Phi^*_{DD|S_{q_2}}\right|^{\left(\frac{\delta^*+|S_{q_2}|+1}{2}\right)}}\times \\
&\quad\quad\frac{\Gamma\left(\frac{\delta+|S_{q_2}|}{2}\right)\Gamma\left(\frac{\delta^*+|S_{q_2}|+1}{2}\right)}{\Gamma\left(\frac{\delta+|S_{q_2}|+1}{2}\right)\Gamma\left(\frac{\delta^*+|S_{q_2}|}{2}\right)}
\end{aligned} \tag{4.8}$$

where $D=\{i,j\}$, $\Phi_{DD|S_{q_2}}=\Phi_{DD}-\Phi_{DS_{q_2}}\left(\Phi_{S_{q_2}S_{q_2}}\right)^{-1}\Phi_{S_{q_2}D}$, and $\Phi_{ii|S_{q_2}}$, $\Phi_{jj|S_{q_2}}$, $\Phi^*_{DD|S_{q_2}}$, $\Phi^*_{ii|S_{q_2}}$ and $\Phi^*_{jj|S_{q_2}}$ are defined similarly.

The next lemma is required to obtain an expression for $h(g',\delta,\Phi)$.

Lemma 4.4.4 *(Lemma 2.13, p. 16, Lauritzen (1996)). Let $\widetilde{C}_1, \ldots, \widetilde{C}_{\widetilde{k}}$ be a perfect sequence with separators $\widetilde{S}_2, \ldots, \widetilde{S}_{\widetilde{k}}$. Assume that $\widetilde{C}_t \subset \widetilde{C}_p$ for some $t \neq p$ and that p is minimal with this property for fixed t. Then*
(a) If $p<t$ then $\widetilde{S}_t=\widetilde{C}_t$ and $\widetilde{C}_1, \ldots, \widetilde{C}_{t-1}, \widetilde{C}_{t+1}, \ldots, \widetilde{C}_{\widetilde{k}}$ is a perfect sequence with separators $\widetilde{S}_2, \ldots, \widetilde{S}_{t-1}, \widetilde{S}_{t+1}, \ldots, \widetilde{S}_{\widetilde{k}}$
(b)If $p>t$ then $\widetilde{S}_p=\widetilde{C}_t$ and $\widetilde{C}_1, \ldots, \widetilde{C}_{t-1}, \widetilde{C}_p, \widetilde{C}_{t+1}, \ldots, \widetilde{C}_{p-1}, \widetilde{C}_{p+1}, \widetilde{C}_{\widetilde{k}}$ is a perfect sequence with separators $\widetilde{S}_2, \ldots, \widetilde{S}_{t-1}, \widetilde{S}_t, \widetilde{S}_{t+1}, \ldots, \widetilde{S}_{p-1}, \widetilde{S}_{p+1}, \widetilde{S}_{\widetilde{k}}$

From Lemma 4.4.4, a perfect sequence of complete sets $\widetilde{C}_1, \ldots, \widetilde{C}_{\widetilde{k}}$ containing the cliques of g' can be thinned by removing complete sets that are not cliques and reordering the sequence. From Lemma 4.4.4, the right-hand side of (4.5) is invariant to this thinning process. Successive application of the thinning process gives a perfect sequence consisting of the cliques of g'.

From (4.5), Theorem 4.4.1 and Lemma 4.4.4, Wong (2002) shows

$$h(g',\delta,\Phi) = \frac{\prod_{i=1,\ldots q-1,q_1,q_2,q+1,\ldots,k}\left[\left|\frac{\Phi_{C_iC_i}}{2}\right|^{\left(\frac{\delta+|C_i|-1}{2}\right)}\Gamma_{|C_i|}\left(\frac{\delta+|C_i|-1}{2}\right)^{-1}\right]}{\prod_{i=2,\ldots q-1,q_1,q_2,q+1,\ldots,k}\left[\left|\frac{\Phi_{S_iS_i}}{2}\right|^{\left(\frac{\delta+|S_i|-1}{2}\right)}\Gamma_{|S_i|}\left(\frac{\delta+|S_i|-1}{2}\right)^{-1}\right]}. \tag{4.9}$$

Now consider the ratio $h(g,\delta,\Phi)/h(g',\delta,\Phi)$. Wong (2002) simplifies the expressions from (4.5) and (4.9) to give

$$\frac{h(g,\delta,\Phi)}{h(g',\delta,\Phi)} = \frac{\left|\Phi_{C_qC_q}\right|^{\left(\frac{\delta+|S_{q_2}|+1}{2}\right)}\left|\Phi_{S_qS_q}\right|^{\left(\frac{\delta+|S_{q_2}|-1}{2}\right)}\Gamma\left(\frac{\delta+\left|S_{q_2}\right|}{2}\right)}{\left|\Phi_{C_{q_1}C_{q_1}}\right|^{\left(\frac{\delta+|S_{q_2}|}{2}\right)}\left|\Phi_{C_{q_2}C_{q_2}}\right|^{\left(\frac{\delta+|S_{q_2}|}{2}\right)}\Gamma\left(\frac{\delta+\left|S_{q_2}\right|+1}{2}\right)2\sqrt{\pi}}. \tag{4.10}$$

Substituting

$$\begin{aligned}\left|\Phi_{C_qC_q}\right| &= \left|\Phi_{DD|S_{q_2}}\right|\left|\Phi_{S_{q_2}}\right| \\ \left|\Phi_{C_{q_1}C_{q_1}}\right| &= \left|\Phi_{ii|S_{q_2}}\right|\left|\Phi_{S_{q_2}}\right| \\ \left|\Phi_{C_{q_2}C_{q_2}}\right| &= \left|\Phi_{ii|S_{q_2}}\right|\left|\Phi_{S_{q_2}}\right|\end{aligned}$$

into (4.10) gives (Wong (2002))

$$\frac{h(g,\delta,\Phi)}{h(g',\delta,\Phi)} = \frac{\left|\Phi_{DD|S_{q_2}}\right|^{\left(\frac{\delta+|S_{q_2}|+1}{2}\right)}\Gamma\left(\frac{\delta+\left|S_{q_2}\right|}{2}\right)}{\left|\Phi_{ii|S_{q_2}}\right|^{\left(\frac{\delta+|S_{q_2}|}{2}\right)}\left|\Phi_{jj|S_{q_2}}\right|^{\left(\frac{\delta+|S_{q_2}|}{2}\right)}\Gamma\left(\frac{\delta+\left|S_{q_2}\right|+1}{2}\right)2\sqrt{\pi}}.$$

A similar expression can be derived for the ratio $h(g,\delta^*,\Phi^*)/h(g',\delta^*,\Phi^*)$ and the result follows.

The following lemma gives an efficient method for evaluating the terms in (4.8) using Cholesky decompositions.

Lemma 4.4.5 *(Wong (2002)). Using the notation of Theorem 4.4.1 and Lemma 4.4.3, suppose that the matrix $A_{C_qC_q} > 0$ is partitioned as*

$$A_{C_qC_q} = \begin{pmatrix} A_{S_{q_2}S_{q_2}} & A_{S_{q_2}D} \\ A_{DS_{q_2}} & A_{DD} \end{pmatrix}$$

and has Cholesky decomposition $A_{C_qC_q} = LL'$ *where*

$$L = \begin{pmatrix} L_{S_{q_2}S_{q_2}} & 0 \\ L_{DS_{q_2}} & L_{DD} \end{pmatrix}$$

and

$$L_{DD} = \begin{pmatrix} l_{\alpha\alpha} & 0 \\ l_{\beta\alpha} & l_{\beta\beta} \end{pmatrix}.$$

Then
(a) $A_{DD|S_{q_2}} = L_{DD}\left(L_{DD}\right)'$
(b) $\left|A_{DD|S_{q_2}}\right| = \left(l_{\alpha\alpha}\right)^2\left(l_{\beta\beta}\right)^2$
(c) $A_{\alpha\alpha|S_{q_2}} = \left(l_{\alpha\alpha}\right)^2$
(d) $A_{\beta\beta|S_{q_2}} = \left(l_{\beta\alpha}\right)^2 + \left(l_{\beta\beta}\right)^2$

Equation (4.18) and parts (b)—(d) of Lemma 4.4.5 give an efficient expression for the conditional distributions in Section 4.8. The main computational effort is in updating the Cholesky decompositions of the matrices $\Phi_{C_qC_q}$ and $\Phi^*_{C_qC_q}$ whenever an edge is added or deleted. From Lemma 4.4.5, this Cholesky decomposition must be done with the entries for the ith and jth vertices in the lower right corner. Note that efficient Cholesky updating routines using Givens rotations are available in MATLAB and FORTRAN. Note also that the dimensions of $\Phi_{C_qC_q}$ and $\Phi^*_{C_qC_q}$ depend on the clique sizes and may be much smaller than p. Thus our method has the local computational properties described in Giudici & Green (1999) and will have similar computational cost to their method per iteration of the Gibbs sampler.

4.5 Prior for Σ

We use the HIW prior of (4.3) for Σ, with $\delta = 5$, because it allows Σ to be integrated out of the sampling scheme described in Section 4.8. Roverato (2000) shows that the inverse of a HIW random matrix has a Wishart distribution, subject to the constraints imposed by the corresponding graph. Thus

$$p(d\Sigma|\Phi,\delta) = \sum_g p(d\Sigma|g,\Phi,\delta)p(g)$$

is a mixture of HIW distributions over all decomposable graphs g, and is equivalent to the prior on Ω being a mixture of constrained Wishart distributions over all decomposable graphs.

4.6 Prior specification for Φ and its parameters

We consider the following three specifications for the hyperparameter Φ, and refer to them as the hyperprior forms of Φ:

1. $\Phi = \tau I,\ \tau > 0$ where I is the $p \times p$ identity matrix.

2. $\Phi = \tau(\rho J + (1-\rho)I),\ \tau > 0$ where J is the $p \times p$ matrix of ones and ρ is a correlation coefficient that needs to be in the open interval $(-1/(p-1), 1)$ for Φ to be positive definite. This specification is used by Giudici & Green (1999) and is called the *equicorrelated version* of Φ because $\Phi_{ii} = \tau$ and $\Phi_{ij} = \tau\rho$ for $i \neq j$.

3. $\Phi = \tau S_y/(n-1)$, where $\tau > 0$,

$$S_y = \sum_{t=1}^{n} (y_t - \overline{y})(y_t - \overline{y})', \tag{4.11}$$

and $\overline{y}$ is the mean of the y_t. We refer to this as the *scaled sum of squares* form of Φ.

We motivate this choice of Φ in two ways. First, by integrating μ out of $p(y|\mu, \Sigma)$, with $p(\mu)$ constant, we obtain

$$p(y|\Sigma) \propto |\Sigma|^{-(n-1)/2} \operatorname{etr}\left(-\frac{1}{2} S_y \Sigma^{-1}\right). \tag{4.12}$$

Suppose g is a decomposable graph. If we take $p(\Sigma|g) \propto p(y|\Sigma)^{1/(n-1)}$, then from (4.12) and equation (3) of Giudici (1996), we can write $p(\Sigma|g)$ in the form (4.4) with $\Phi = S_y/(n-1)$.

A second motivation for this choice of Φ is to note that if $\Sigma \sim HIW(p, \delta, \Phi)$, then $E(\Sigma_{CC}) = \Phi_{CC}/(\delta - |C| - 1)$ for any clique $C = C_i$ or separator $C = S_i$ in (4.3). Since $(S_y)_{CC}/(n-1)$ is an unbiased estimator of Σ_{CC}, this suggests taking $\Phi \propto S_y/(n-1)$.

We assume in all cases that τ is uniform on the interval $[0, \Gamma]$ where Γ is large, e.g. $\Gamma = 10^{10}$, and in the equicorrelated case that ρ is uniform on the open interval $(-1/(p-1), 1)$.

4.7 Prior for g

Because of the theoretical and practical difficulty in calculating the exact number of decomposable graphs for a given p, or the number of graphs of a given size, most of the literature for both decomposable and general models takes the prior for g as uniform over all the relevant graphs; see, for example, Atay-Kayis & Massam (2005), Dellaportas & Forster (1999), Geiger & Heckerman (2002), Giudici & Castelo (2003), Brooks et al. (2003), Giudici & Green (1999), Roverato (2002).

Such a prior favours any class of graphs with many members over a class with few members, and favours middle sized graphs over both very large and very small sized graphs.

Let $A_{p,k}$ denote the number of graphs of size k. We specify the prior for a graph g hierarchically as follows.

$$p(g|size(g) = k) = \frac{1}{A_{p,k}},$$

so that all graphs of a given size are equally likely. We now specify the prior for the size of a graph. One choice is

$$p(size = k) \propto A_{p,k},$$

which means that

$$p(g) = p(g, size(g)) = p(g|size(g))p(size(g)) \propto constant,$$

giving the uniform prior for g. A more flexible prior is of the form

$$p(size = k|\psi) = \binom{r}{k}\psi^k(1-\psi)^{r-k}$$

where we interpret ψ as the probability that any two vertices have a common edge and $r = 2^{p(p-1)/2}$ is the maximum possible number of edges. We could then put a

prior on ψ. Suppose we take the prior for ψ to be a Beta distribution function with parameters a and b, i.e.

$$p(\psi) = \frac{\psi^{a-1}(1-\psi)^{b-1}}{B(a,b)}.$$

Then,

$$p(size = k) = \binom{r}{k} \frac{B(a+k, r-k+b)}{B(a,b)}$$

and

$$p(g) = p(g|size(g))p(size(g)) \tag{4.13}$$

$$= \binom{r}{size(g)} \frac{B(a+size(g), r-size(g)+b)}{A_{p,size(g)}B(a,b)} \tag{4.14}$$

where $B(a,b)$ is the beta function. We can also put a prior on a, b. We take ψ uniform so that $a = b = 1$, which means that

$$p(size = k) = \frac{1}{(r+1)} \text{ and } p(g) = \frac{1}{(r+1)A_{p,k}} .$$

That is, the size of each graph has equal probability, and the probability of a graph of size k conditional on $size = k$ is uniform. However our framework is more flexible than this.

We call this the *size-based prior* for g and in Section 7.2 compare results against those using a uniform prior.

The size-based prior makes it easier to discover sparse and full graphs when n/p is small. The counts $A_{p,k}$ are not available in the literature. Section 7.3 gives results to calculate a subset of them analytically, and shows how to evaluate the rest by simulation.

4.8 Posterior inference and Markov chain Monte Carlo sampling

We use Markov Chain Monte Carlo (MCMC) simulation to obtain all posterior distributions. The simulation involves the generation of the graphs g and the parameters in Φ but not Σ and μ which are integrated out. Thus, our sampling scheme is said to generate from *reduced conditionals* and is therefore far more efficient than the

sampling schemes in Giudici & Green (1999) and Wong et al. (2003) that generate Σ as part of their sampling scheme.

We note that iterates of μ and Σ can also be generated in conjunction with the simulation, but such iterates of μ and Σ do not have any influence on the convergence properties or dependence structure of the reduced conditional simulation.

The following theorems are useful in evaluating the conditional distributions required in the simulations. The first theorem gives a conjugate prior property of the HIW distribution.

Let S_y be defined by (4.11) and define

$$\Phi^* = \Phi + S_y \text{ and } \delta^* = \delta + n^* \tag{4.15}$$

where $n^* = n$ if the mean μ is known and $n^* = n - 1$ otherwise.

Theorem 4.8.1 *(Dawid and Lauritzen, 1993) For the Bayesian model specified by (4.3) and (4.12)*

$$\Sigma | y, \delta, \Phi, g \sim HIW(g, \delta^*, \Phi^*)$$

Proof. See Dawid and Lauritzen (1993), Wong (2002), or Appendix 8.3. ■

The next theorem gives an expression for the marginal likelihood.

Theorem 4.8.2 *(Giudici, 1996) For the Bayesian model specified by (4.3) and (4.12),*

$$p(y|\delta, \Phi, g) = (2\pi)^{-(n^* p/2)} \frac{h(g, \delta, \Phi)}{h(g, \delta^*, \Phi^*)} \tag{4.16}$$

Proof. See Giudici (1996), Wong (2002), or Appendix 8.3. ■

4.9 Sampling the graphs g

We sample the graphs g by generating the edge indicators one at a time, conditional on δ, Φ and $e_{-ij} = \{e_{kl}, (k,l) \neq (i,j), k < l\}$ using the following MH sampling scheme.

Using the notation of Section 4, let $g^c = (V, E^c)$ be the current graph of Σ, which is decomposable by construction with edge indicators $\{e^c_{kl} : \ 1 \leq k < l \leq p\}$.

We choose a pair (i, j) at random and suppose that $g = (e_{ij}, e^c_{-ij})$ is decomposable for both $e_{ij} = 0$ and $e_{ij} = 1$. We use the legal edge addition and deletion characterizations of Giudici & Green (1999) and Frydenberg & Lauritzen (1989) respectively to ensure this. Otherwise we choose a new pair (i, j).

Set the proposal graph as g^p (conditional on g^c) as $g = (e^p_{ij}, e^c_{-ij})$ where $e^p_{ij} = 1 - e^c_{ij}$. This means that the proposal density for e_{ij} is $q_g(a|b, e^c_{-ij})$ where a and b are each either 0 or 1, and $q_g(a = 1 - b|b, e^c_{-ij}) = 1$.

The MH acceptance probabilty for the proposal is

$$\min\left\{1, \frac{p(y|g^p, \Phi, \delta)}{p(y|g^c, \Phi, \delta)} \frac{p(g^p)}{p(g^c)}\right\} \tag{4.17}$$

because $q_g(e^c_{ij}|e^p_{ij}, e^c_{-ij})/q_g(e^p_{ij}|e^c_{ij}, e^c_{-ij}) = 1$. The ratio $p(g^p)/p(g^c)$ is known and the ratio of marginal likelihoods is

$$\frac{p(y|g^p, \Phi, \delta)}{p(y|g^c, \Phi, \delta)} = \frac{h(g^p, \delta, \Phi)}{h(g^c, \delta, \Phi)} \frac{h(g^c, \delta^*, \Phi^*)}{h(g^p, \delta^*, \Phi^*)}. \tag{4.18}$$

A simple expression for (4.18) is derived in Section 4.4.

4.10 Generating the parameters in Φ

In all cases of Section 4.6 we generate τ using a random walk MH method

$$\log(\tau^p) = \log(\tau^c) + \xi_\tau \ , \quad \xi_\tau \sim N(0, \sigma^2_\tau),$$

which has acceptance probability

$$\min\left\{1, \frac{p(y|g, \tau^p, \rho)}{p(y|g, \tau^c, \rho)} \frac{p(\tau^p)}{p(\tau^c)}\right\} \tag{4.19}$$

as the proposal densities cancel out. In the equicorrelated case, the parameter ρ is generated similarly to τ by a random walk MH method

$$\rho^p = \rho^c + \xi_\rho \ , \quad \xi_\rho \sim N(0, \sigma^2_\rho).$$

The choice of the variances $\sigma^2_\tau, \sigma^2_\rho$ is sensitive to p, and is fine tuned to attain acceptance probabilities of around 25% according to the acceptance rate of the proposals. For the cases $p = 17$ reported in this thesis, such an acceptance probability resulted from using $\sigma^2_\tau = 1/10$ and $\sigma^2_\rho = 1/20$.

4.11 Generating Σ, Ω and μ

Although μ, Σ and Ω are not generated in the Markov chain Monte Carlo simulation, it is often necessary to estimate functionals of μ, Σ and Ω. Such functionals can be estimated by sampling from the posterior distribution of Σ, Ω and μ. Conditional on (g, δ, Φ) it follows from Theorem 4.8.1 that $p(\Sigma|y, g, \delta, \Phi)$ is HIW with parameters $(\delta + n - 1, \Phi + S_y)$ so that Σ and Ω can be generated using Theorems 3 and 4 of Roverato (2000). It is straightforward to show that $p(\mu|y, \Sigma, g, \delta, \Phi)$ is $N(\overline{y}, \Sigma/n)$, and hence generate μ, giving iterates $\{\mu^{[j]}, \Sigma^{[j]}, \Omega^{[j]},\ j \geq 1\}$ from the posterior distribution.

4.12 Generating δ

We now show how to generate δ assuming bounds $d_L \leq \delta \leq d_U$. Define the proposal density $q_\delta(x|z)$ as

$$q_\delta(x|z) = \begin{cases} \frac{1}{2}, & \text{if } x = z+1 \text{ or } x = z-1 \text{ for } d_L \leq z \leq d_U; \\ 1, & \text{if } x = z+1 \text{ for } z = d_L; \\ 1, & \text{if } x = z-1 \text{ for } z = d_U; \\ 0, & \text{otherwise.} \end{cases} \tag{4.20}$$

$$q_\delta(x = z+1|z) = \frac{1}{2} = q_\delta(x = z-1|z)$$

for $d_L \leq z \leq d_U$, and

Now use q_δ as a MH proposal for generating δ with acceptance probability

$$\min\left\{1, \frac{p(y|g, \delta^p, \Phi)}{p(y|g, \delta^c, \Phi)} \frac{p(\delta^p)}{p(\delta^c)} \frac{q_\delta(\delta^c|\delta^p)}{q_\delta(\delta^p|\delta^c)}\right\} \tag{4.21}$$

We take $p(\delta)$ as the prior for δ. This can be uniform on the closed interval $[d_L, d_U]$. Note that

$$q_\delta(\delta^p|\delta^c) = \begin{cases} \frac{1}{2}, & \text{if } d_L \leq \delta^c \leq d_U; \\ 1, & \text{if } \delta^c = d_L \text{ or } d_U. \end{cases} \tag{4.22}$$

and

$$q_\delta(\delta^p|\delta^c) = \begin{cases} \frac{1}{2}, & \text{if } d_L \leq \delta^p \leq d_U; \\ 1, & \text{if } \delta^p = d_L \text{ or } d_U. \end{cases} \tag{4.23}$$

Note also that if $d_L = d_U$ then there is nothing to generate, and $d_U - d_L$ can be as large or small as you want.

4.13 Efficient estimation of $E(\Omega|y)$

This section shows how to estimate the posterior mean of Ω efficiently. The posterior mean of Ω is not only used as an estimator of Ω, but $E(\Omega|y)^{-1}$ is the Bayes estimator of Σ for the L_1 loss function used in Section 7.2. One method of estimating $E(\Omega|y)$ is to use the histogram estimator $J^{-1}\sum_{j=1}^{J}\Omega^{[j]}$. A statistically more efficient estimator is the mixture estimator $J^{-1}\sum_{j=1}^{J}E(\Omega|y,g^{[j]},\delta^{[j]},\Phi^{[j]})$. We now show how to efficiently compute $E(\Omega|y,g,\delta,\Phi)$ using the following notation from Lauritzen (1996). Suppose that A is a $p\times p$ matrix and $S\subset V$. Let $B=[A_{SS}]^V$, the $p\times p$ matrix defined by

$$B_{ij}=\begin{cases} A_{ij} & \text{if } \{i,j\}\subset S \\ 0 & \text{otherwise}\end{cases}$$

Theorem 4.13.1 *Suppose that $\Omega|y\sim W(g,\delta^*,\Phi^*)$, where g is decomposable. Then, using the notation of this and Section 4,*

$$E\left(\Omega|y,\delta,\Phi,g\right)=\sum_{i=1}^{k}\left[\left(\delta^*+|C_i|-1\right)\left(\Phi^*_{C_iC_i}\right)^{-1}\right]^V-\sum_{i=2}^{k}\left[\left(\delta^*+|S_i|-1\right)\left(\Phi^*_{S_iS_i}\right)^{-1}\right]^V \tag{4.24}$$

Proof. See Wong (2002) or Appendix 8.3. ■

4.14 Comparsion to the Wong et al. (2003) covariance selection prior

This section compares the performance of our prior to the covariance selection prior of Wong et al. (2003), which does not assume that the graph of the covariance matrix is decomposable. Based on the results in Section 7.2, we use the equicorrelated form of Φ and the size-based prior for the decomposable graphs.

The simulation considers the following four graph types for g. (a) $\Omega=I$, the identity matrix, representing the empty graph and a diagonal covariance matrix;

(b) Ω tridiagonal, representing a sparse and decomposable graph which is a path consisting of $p-1$ edges; (c) Ω corresponding to a 4-cycle on p vertices representing a sparse but nondecomposable graph; and (d) Ω corresponding to a $p-$cycle on p vertices, again representing a sparse but nondecomposable graph. We note that the nondecomposable graphs in (c) and (d) require the addition of extra edges when we estimate them by a mixture of decomposable graphs. Furthermore, (d) is an extreme case of non-decomposability, as it requires the addition of at least $p-3$ edges to make the graph decomposable. Conversely, the unchorded 4-cycle on p vertices requires the addition of only one edge to make it decomposable, so it is chosen as an indicator of performance for the sparsest nondecomposable case.

The simulation considers the three forms of Φ described in Section 4.6 and two sample sizes $n = 40$ and $n = 100$. We report results for matrices of size $p = 17$, but similar results are obtained for matrices of other sizes.

Let Σ_T be the true value of Σ and let $\widehat{\Sigma}$ be an estimator of Σ_T. We measure the performance of $\widehat{\Sigma}$ using the L_1 loss function

$$L_1(\widehat{\Sigma}, \Sigma_T) = \text{trace}(\widehat{\Sigma}\Sigma_T^{-1}) - \log\det(\widehat{\Sigma}\Sigma_T^{-1}) - p. \tag{4.25}$$

This loss function is frequently used to compare estimates of the covariance matrix, e.g. Yang & Berger (1994). It is straightforward to show that $L_1 \geq 0$ for all $\widehat{\Sigma}$ and Σ_T, and that it is only equal to 0 if $\widehat{\Sigma} = \Sigma_T$. It is also straightforward to show that for $y \sim N(0, \Sigma)$,

$$L_1(\widehat{\Sigma}, \Sigma_T) = -\int p(y|\widehat{\Sigma}) \log\left(\frac{p(y|\Sigma_T)}{p(y|\widehat{\Sigma})}\right) dy \tag{4.26}$$

i.e. L_1 is equivalent to a Kullback-Liebler distance between $p(y|\Sigma_T)$ and $p(y|\widehat{\Sigma})$ with respect to the density $p(y|\widehat{\Sigma})$. The Bayes estimator for Σ for the L_1 loss function is $E(\Omega|y)^{-1}$, which is computed as in Section 4.13.

We refer to the decomposable prior as DCP and the nondecomposable prior of Wong et al. (2003) as NDP. We use boxplots to compare replication by replication the DCP prior with the NDP prior in terms of the percentage *increase* in the loss function L_1 resulting from using the NDP prior compared to the DCP prior. That is, the boxplots are based on calculating the percentage increase in L_1 of DCP over NDP for each iterate, i.e.

$$100(L_1^{DCP} - L_1^{NDP})/L_1^{NDP}.$$

for each replication, where L_1^{DCP} and L_1^{NDP} are the values of $L_1(\widehat{\Sigma}, \Sigma_T)$ for the DCP and NDP priors, respectively.

We ran the sampler for the case $p = 17$ on $n = 40$ and 100 observations from four simulated data sets corresponding to the four models (a)–(d) for Ω.

Figure 4.3 reports boxplots of the percentage increase in L_1 of DCP over NDP for each iterate. The boxplots are based on 20 replications with each replication consisting of 2,000 burn-in iterations and 20,000 sampling iterations. This number of iterations was sufficient because Ω is relatively sparse in the four models considered. Figure 4.3 shows that both priors perform similarly for decomposable graphs and nondecomposable graphs, for both $n = 40$ and $n = 100$. These results and others suggest that the prior based on decomposable graphs performs similarly to that of Wong et al. (2003) when the graphs are relatively sparse.

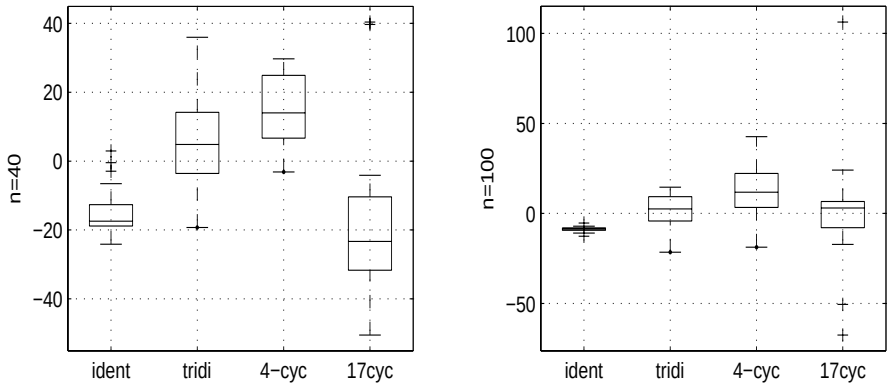


Figure 4.3: Percentage increase in L_1 for DCP over NDP. The left panel is for $n = 40$ and the right is for $n = 100$.

Next we report autocorrelation plots for the iterates of the elements of Ω, when $p = 5$ and the graph is full for both DCP and NDP when $n = 40$. The simulation for DCP uses a burn-in of 50,000 iterations and a sampling of 50,000 iterations, and $500,000$ burn-in and 1 million sampling iterations for NDP, because Ω is not sparse.

Figures 4.4 and Figure 4.5 show the autocorrelation plots for the DCP and NDP models for a representative selection of Ω_{ij}. Figures 4.4 and 4.5 show that the autocorrelations of the iterates of the Ω_{ij} decay rapidly to zero for the DCP model, but are far more dependent in the NDP model. This difference in dependence is due to the greater efficiency of the sampling scheme in the decomposable case. Grey

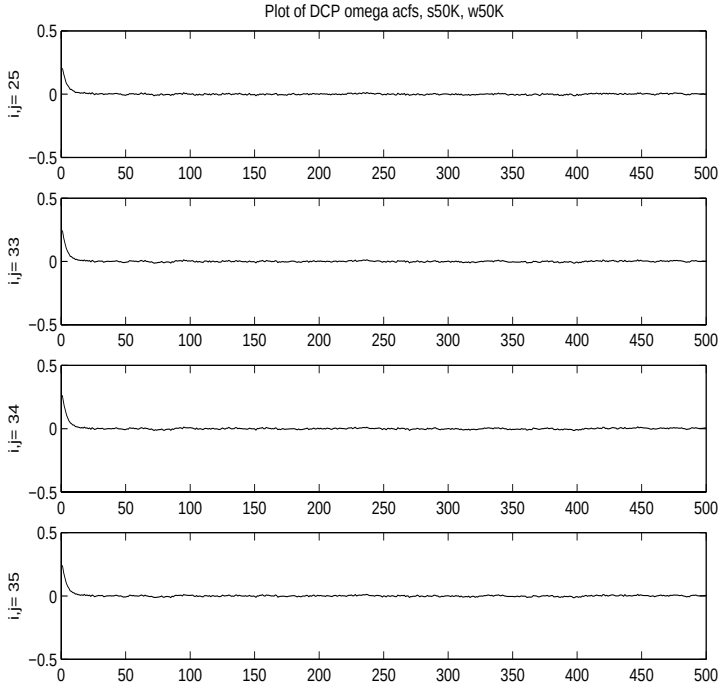


Figure 4.4: Autocorrelations of the iterates of the Ω_{ij} in the DCP case for a representative selection of Ω_{ij}.

scale plots of the true inverse covariance Ω and posterior mean estimates of Ω for the NDP estimator and the DCP estimator for the 17-cycle case indicate that NDP and DCP performed similarly in the simulations. For brevity only the nondecomposable 17-cycle is presented as it represents a case of high non-decomposability. Figure 4.6 shows that even in this case, the grey scales are very similar.

As a final empirical check on how close the DCP model estimate and NDP model estimates were, the average value of all entries $\widehat{\Omega}_{ij}$ from 20 DCP model estimates $\widehat{\Omega}$ are compared to the same for the NDP model estimate. In the case of Figure 4.6, they are within 2.5% of each other.

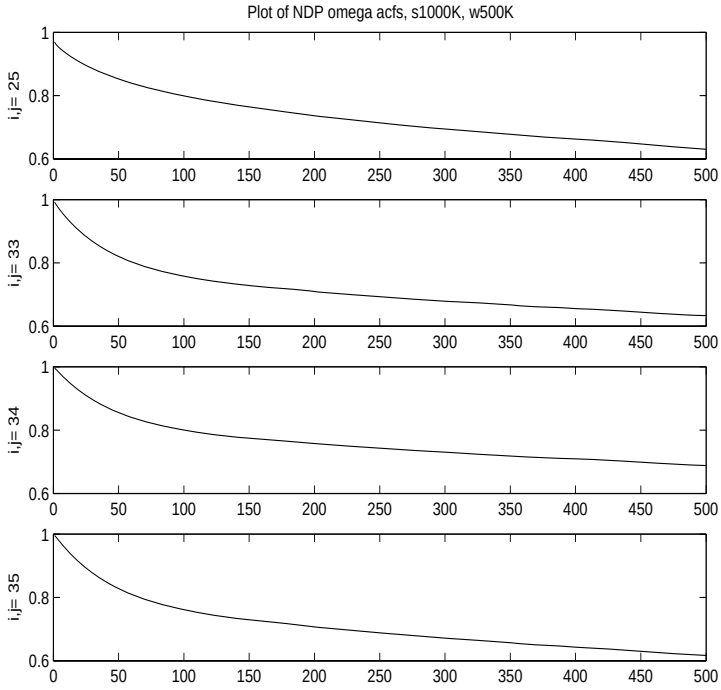


Figure 4.5: Autocorrelations of the iterates of the Ω_{ij} in the NDP case for a representative selection of Ω_{ij}.

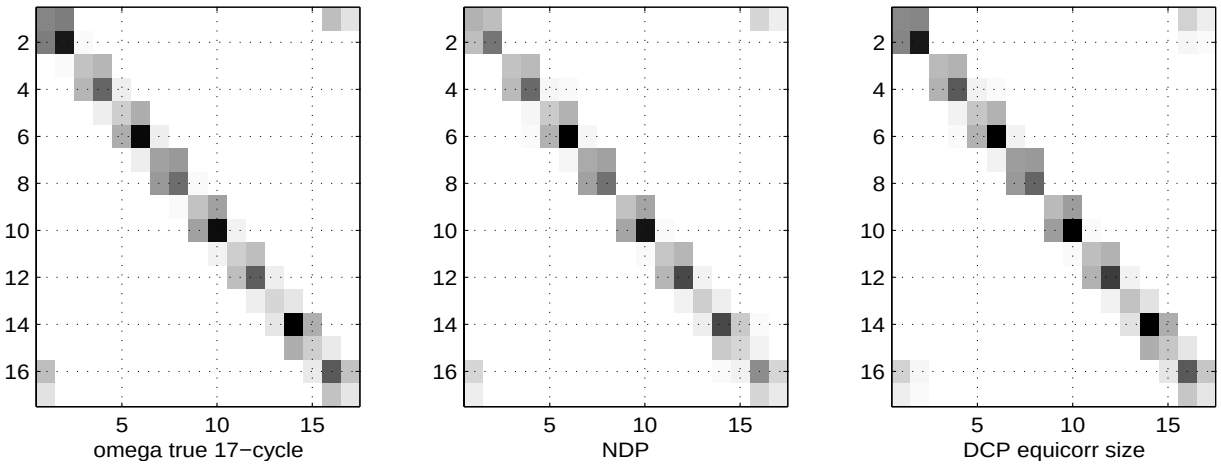


Figure 4.6: True inverse covariance Ω and posterior mean estimates of Ω for the NDP estimator and the DCP estimator for the 17-cycle case.

Chapter 5

Variable and covariance selection in multivariate regression models

5.1 Introduction

Decomposable graphical models have a natural application in general linear regression. This chapter provides a decomposable graphical framework for both deciding the structure of, and estimating the parameters in, a multivariate regression model. It compares the results to those obtained by Cripps et al. (2005) for the same real data sets, who use the prior of Wong et al. (2003) to carry out covariance selection.

By variable selection we mean that the regression model allows some of the regression coefficients to be identically zero. By covariance selection we mean that the model allows some of the off-diagonal elements of the inverse of the covariance matrix to be identically zero. We estimate all functionals of the parameters by model averaging; that is, by taking a weighted average of the values of the functional, where the average is over the allowable configurations of the regression coefficients and the covariance matrix, and the weights are the posterior probabilities of the configurations. The computation is carried out using a Markov chain Monte Carlo simulation method.

This chapter is organised as follows. Section 5.2 describes the multivariate model and the priors for variable and covariance selection. Section 5.3 discusses the sampling scheme and how the decomposable reduced conditional sampler carries out the computation. Section 5.4 presents the decomposable results and shows that they are

very similar to those obtained by Cripps et al. (2005), but that the autocorrelations in the iterates of the decomposable sampling scheme decay far more rapidly to zero than the iterates in the Cripps et al. (2005) sampling scheme.

5.2 Model description

5.2.1 Introduction

For $t = 1, \ldots, n$, let y_t be a $p \times 1$ vector of responses, x_t a $p \times q$ matrix of covariates and β the $q \times 1$ vector of regression coefficients. We assume the model

$$y_t = x_t\beta + e_t, \quad e_t \sim N(0, \Sigma), \tag{5.1}$$

where g is decomposable and $\Sigma \sim HIW(g, \delta, \Phi)$. Let $\gamma = (\gamma_1, \ldots, \gamma_q)$ be a vector of binary variables such that the ith column of x_t is included in the regression if $\gamma_i = 1$ and excluded if $\gamma_i = 0$. We write $x_{t,\gamma}$ for the matrix that contains all columns of x_t for which $\gamma_i = 1$ and write β_γ for the corresponding subvector of regression coefficients. Therefore, the vector γ indexes all the mean functions for the regression model (5.1). Conditional on γ, (5.1) becomes

$$y_t = x_{t,\gamma}\beta_\gamma + e_t, \quad e_t \sim N(0, \Sigma), \quad \Sigma \sim HIW(g, \delta, \Phi). \tag{5.2}$$

Model (5.2) contains as a special case the multivariate model

$$y_t = B\tilde{x}_t + e_t, \quad e_t \sim N(0, \Sigma), \quad \Sigma \sim HIW(g, \delta, \Phi), \tag{5.3}$$

where B is a matrix of regression coefficients and x_t is a vector of covariates. It is clear that model (5.3) is a special case of model (5.1) by taking $x_t = \tilde{x}_t' \otimes I_p$ and $\beta = vec(B)$, where $\otimes$ means Kronecker product and $vec(B)$ is the vector obtained by stacking the columns of B beneath each other. The model (5.3) is used extensively in multivariate regression analysis, e.g. , (Mardia et al., 1979,p. 157) and in particular Brown et al. (1998), Brown et al. (1999) and Brown et al. (2002). We note that Brown et al. (1998), Brown et al. (1999) and Brown et al. (2002) do variable selection on $\tilde{x}_t$ which means that when they drop a covariate they drop a whole column of the matrix B. (Cripps et al., 2005,Section 2.8) shows how this can be done in general for the model (5.1).

5.2.2 Prior for the regression coefficients

We use the same prior for the regression coefficients as in Cripps et al. (2005). The following description is taken directly from Cripps et al. (2005) with the kind permission of the authors, and included here only for completeness.

Similarly to Smith & Kohn (1996), we define the prior for the the regression coefficients as being noninformative with respect to the likelihood and with a mode at zero. To motivate the prior, it is useful to rewrite the likelihood as follows.

$$\begin{aligned} p(y|\beta,\gamma,\Omega) &= |2\pi\Omega|^{\frac{n}{2}}\exp\left\{-\frac{1}{2}\sum_{t=1}^{n}(y_t - x_{t,\gamma}\beta_\gamma)'\Omega(y_t - x_{t,\gamma}\beta_\gamma)\right\} \\ &= |2\pi\Omega|^{\frac{n}{2}}\exp\left\{-\frac{1}{2}\sum_{t=1}^{n}y_t'\Omega y_t - 2\beta_\gamma'\sum_{t=1}^{n}x_{t,\gamma}'\Omega y_t + \beta_\gamma'\sum_{t=1}^{n}x_{t,\gamma}'\Omega x_{t,\gamma}\right\} \\ &= |2\pi\Omega|^{\frac{n}{2}}\exp\left\{-\frac{1}{2}SSy - 2\beta_\gamma' SSxy_\gamma + \beta_\gamma' SSxx_\gamma\beta_\gamma\right\}, \end{aligned} \tag{5.4}$$

where

$$SSy = \sum_{t=1}^{n}y_t'\Omega y_t, \quad SSxy_\gamma = \sum_{t=1}^{n}x_{t,\gamma}'\Omega y_t, \quad \text{and} \quad SSxx_\gamma = \sum_{t=1}^{n}x_{t,\gamma}'\Omega x_{t,\gamma}.$$

As a function of β_γ, the likelihood is Gaussian with a mean of $(SSxx_\gamma)^{-1}SSxy_\gamma$ and covariance matrix $(SSxx_\gamma)^{-1}$.

Conditional on the binary indicator vector and the covariance matrix we take the prior for β_γ as

$$\beta_\gamma|\Sigma,\gamma \sim N(0, c(SSxx_\gamma)^{-1}) \tag{5.5}$$

and set $c = n$ such that the prior variance of β_γ stays approximately the same as n increases.

From (5.4) and (5.5) we can write the density of β_γ conditional on $y, \Sigma, \gamma, g, \delta$ and Φ as

$$\beta_\gamma|y,\gamma,\Sigma \sim N\left(\frac{c}{1+c}(SSxx_\gamma)^{-1}SSxy_\gamma, \frac{c}{1+c}(SSxx_\gamma)^{-1}\right), \tag{5.6}$$

where $\Sigma \sim HIW(g,\delta,\Phi)$.

5.2.3 Prior for the vector of binary indicator variables

The prior for the vector of binary indicator variables is that of Cripps et al. (2005). The following description is taken directly from Cripps et al. (2005) with the kind permission of the authors, and included here only for completeness.

We first define

$$q_\gamma = \sum_{i=1}^{q} \gamma_i,$$

which is the number of columns contained in x_t specified by $\gamma_i = 1$.

We assume that γ and Σ are independent apriori and as in Kohn et al. (2001) we specify the prior for γ as

$$p(\gamma|\pi) = \pi^{q_\gamma}(1-\pi)^{q-q_\gamma}, \qquad \text{with} \qquad 0 \leq \pi \leq 1. \tag{5.7}$$

We set the prior for π as uniform, i.e. $p(\pi) = 1$ for $0 \leq \pi \leq 1$, so that

$$\begin{aligned} p(\gamma) &= \int p(\gamma|\pi)p(\pi)d\pi \\ &= \int \pi^{q_\gamma}(1-\pi)^{q-q_\gamma}d\pi \\ &= \mathrm{B}(q_\gamma + 1, q - q_\gamma + 1) \end{aligned}$$

where B is the beta function defined by

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

The likelihood for γ and Σ with β_γ integrated out is

$$\begin{aligned} p(y|\gamma,\Sigma) &= \int p(y|\beta,\Sigma,\gamma)p(\beta_\gamma|\gamma,\Sigma)d\beta_\gamma \\ &\propto (1+c)^{-\frac{q_\gamma}{2}}\exp\left\{-\frac{1}{2}\left(SSy - \frac{c}{1+c}SSxy_\gamma' SSx x_\gamma^{-1} SSxy_\gamma\right)\right\}. \end{aligned} \tag{5.8}$$

We can write the density of γ conditional on y and Σ as

$$p(\gamma|y,\Sigma) \propto p(y|\Sigma,\gamma)p(\gamma),$$

and use this density to update γ in the Markov chain Monte Carlo simulation.

5.2.4 Permanently selected variables

We frequently wish to permanently retain some variables in the regression. For example, we may wish to retain all the intercept terms in the regression. We do so by setting the indicators γ for these variables to be identically one and setting q_γ in section 5.2.3 to be the sum of the γ_i, excluding those γ_i that are identically 1. If we wish to estimate the model with no variable selection, then we would set $\gamma_i = 1$ for all i.

5.2.5 Priors for Σ, Φ, g

We use the HIW prior of (4.3) for Σ, with $\delta = 5$, because it allows Σ to be integrated out in the sampling scheme described in Section 4.8. Based on the results in Section 7.2, we use the equicorrelated form of Φ and the size-based prior for the decomposable graphs.

5.3 Sampling scheme

This section describes the decomposable sampling scheme for the regression model. The sampling scheme for γ and β in this chapter is the same as the sampling scheme for γ and β in Cripps et al. (2005), with the added condition that $\Sigma \sim HIW(g, \delta, \Phi)$ is decomposable. Conditional on $\Sigma \sim HIW(g, \delta, \Phi)$, we do variable selection on the elements of β, with β integrated out. To do this, we use a Gibbs sampler to generate the elements of γ one at a time by calculating $p(\gamma_i = 1|y, \Omega, \gamma_{-i})$, where $\gamma_{-i} = \{\gamma_j : j \neq i\}$ (see Kohn et al. (2001) for details). In order to estimate the mean for covariance selection, we then generate β_γ conditional on Σ, γ from its conditional posterior density as given in (5.6). By conditioning on Σ we are also conditioning on g, δ, Φ because $\Sigma \sim HIW(g, \delta, \Phi)$.

The sampling scheme for covariance selection as described in this chapter is different to Cripps et al. (2005) who do covariance selection on the elements of Ω, one at a time, conditional on β and Ω. In the decomposable scheme, on the other hand, covariance selection for Ω is performed by generating the decomposable graph g, via the edge indicators, as described in Section 4.9. This is done with Ω integrated out but dependent on β because $\tilde{y} = y - x\beta$ is used to calculate S_y.

In order to compare performance and use $\tilde{y} = y - x\beta$ to calculate S_y, we generate Ω. Ω is sampled directly from a decomposable HIW distribution using the results of Sections 4.3 and 4.11 where $\tilde{y}$ is used to calculate S_y.

In summary, γ, β, g, Ω and the parameters in Φ are generated using the following Markov chain Monte Carlo scheme.

1. $\gamma_i|\gamma_{-i}, y, g, \Omega$, $i = 1, \ldots q$;
2. $\beta_\gamma|y, \gamma, g, \Omega$;
3. $\rho, \tau|y, g, x, \beta, \delta, \Phi$;
4. $e_{ij}|e_{-ij}, y, x, \beta, \delta, \Phi$, $i = 1, \ldots p-1$ $j = i, \ldots p$;
5. $\Omega|y, x, \beta, g, \delta, \Phi$,

where the edge indicators e_{ij} and the notation $g = (e_{ij}, e_{-ij})$ are given in Section 4.4. We generate $g = (e_{ij}, e_{-ij})$ and the parameters in Φ using the sampling scheme described in Chapter 4.8. Ω is sampled from its posterior distribution, conditional on $x\beta, g, \delta, \Phi$, using the method and results described in Sections 4.3 and 4.11.

5.4 Comparison to Cripps et al. (2005) using same real data sets

This section compares the performance of the decomposable prior to the covariance selection prior of Wong et al. (2003) in the multivariate regression context described in Cripps et al. (2005), which does not assume that the graph of the covariance matrix is decomposable. Based on the results in Section 7.2, we use the size-based prior for the decomposable graphs. We use the equicorrelated form of Φ when it is most likely that Ω is sparse, and both the equicorrelated and scaled sum of squares forms of Φ otherwise.

We compare performance of the priors on the cross-sectional cow diet and pigs growth rate datasets reported in Cripps et al. (2005) and explained in detail in Subsections 5.4.2 and 5.4.1, as well as the physical measurements dataset from Larner (1996). Following the notation of Cripps et al. (2005) and Section 4.14,

let $DCPCSVS$ be the decomposable prior and let $NDPCSVS$ be the nondecomposable prior of Cripps et al. (2005). Let τS_y stand for the scaled sum of squares form of Φ for model $DCSVS$. Let C be the partial correlations matrix. For $j = 1, \ldots, p$ and $i < j$, we define the binary variable $J_{ij} = 0$ if C_{ij} is identically zero and $J_{ij} = 1$ otherwise. Let $J = \{J_{ij}, i < j, j = 1, \ldots, p\}$. These binary variables are analogous to the γ_i binary variables that we use for variable selection, and in the decomposable case, J_{ij} are the edge indicators e_{ij}.

For the variable selection, we report the posterior means and standard errors of the regression coefficients, and the posterior probabilities of including a predictor variable in the regression. For the covariance selection, we report image plots of the estimates of C, J and Ω which are computed as the average of the iterates and which we we refer to as $\widehat{C}, \widehat{J}$ and $\widehat{\Omega}$, respectively. The image plots are lighter where the matrix is sparser. Where instructive, we report the graphs consisting of edges with at least $k\%$ posterior sampling probability, and refer to these as the $k\%$ graphs. Note that the decomposable $k\%$ graphs need not be decomposable.

5.4.1 Pig growth rate data

This longitudinal data set contains observations on 48 pigs measured over 9 successive weeks. It is described in Diggle et al. (2002,pp. 34-35). [1] Following Cripps et al. (2005), we model the mean function of the pigs growth rate as a piecewise linear trend such that at each time point we allow the slope to change. We use the equicorrelated form of Φ because longitudinal data is most likely to result from a tridiagonal banded structure. Therefore a sparse structure for Φ is likely to give the best results.

Let y_{ti} be the response for pig t at time i and write the piecewise linear time trend as

$$\beta_0 + \beta_1 i + \beta_2 (i-2)_+ + \beta_3 (i-3)_+ + \ldots + \beta_8 (i-8)_+, \text{ for } i = 1, \ldots, 9 \qquad (5.9)$$

where

$$x_+ = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases}$$

Writing (5.9) in the notation of (5.3)

$$y_t = x_t \beta + e_t, \qquad e_t \sim N(0, \Sigma)\,, \qquad (5.10)$$

[1] The data can be obtained from http://www.maths.lancs.ac.uk/∼diggle/lda/Datasets/

where the predictor matrix x_t is

$$X_t = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 5 & 3 & 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 6 & 4 & 3 & 2 & 1 & 0 & 0 & 0 \\ 1 & 7 & 5 & 4 & 3 & 2 & 1 & 0 & 0 \\ 1 & 8 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ 1 & 9 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

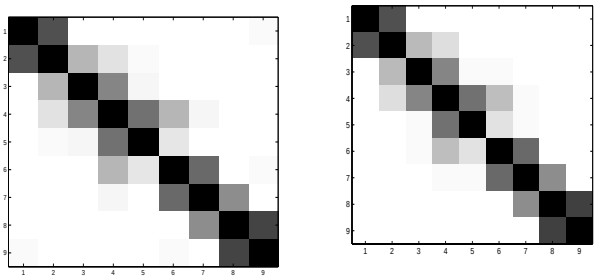
and β is the corresponding 9×1 vector of regression coefficients.

Table 5.1 compares the estimated posterior means and standard errors for the regression coefficients and the posterior probabilities that the regression coefficients are nonzero. The coefficient β_2 is significant for both $NDPCSVS$ and $DCPCSVS$. Comparing respective entries indicates that there is no significant difference in the results obtained by $NDPCSVS$ and $DCPCSVS$.

	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9
Post. mean $NDPCSVS$	17.8010	6.6829	0.1251	-1.2903	0.1648	0.3164	-0.0604	0.6258	-0.7225
Post. mean $DCPCSVS$	17.8092	6.6861	0.1149	-1.2779	0.1647	0.3064	-0.0674	0.6609	-0.7596
Post. std. error $NDPCSVS$	0.3903	0.1630	0.2238	0.2920	0.2357	0.3759	0.2233	0.4031	0.4614
Post. std. error $DCPCSVS$	0.4079	0.1603	0.2218	0.2980	0.2321	0.3684	0.2269	0.3846	0.4120
Post. prob. $NDPCSVS$	NA	1.0000	0.3950	1.0000	0.5185	0.5905	0.3465	0.8180	0.8145
Post. prob. $DCPCSVS$	NA	1.0000	0.3889	0.9998	0.5141	0.5843	0.3484	0.8367	0.8609

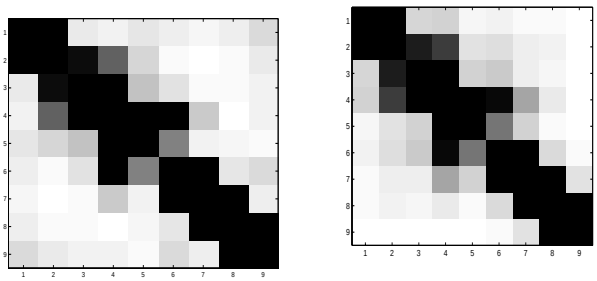
Table 5.1: Posterior means, standard errors and probabilities of being nonzerofor the regression coefficients using model $NDPCSVS$ for the pig growth rate data compared to the same estimates using model $DCPCSVS$. NA means not applicable as the coefficient is always included.

Figure 5.1 compares the image plots of $\widehat{C}$ and suggests that the estimates are similar. Figure 5.2 compares the image plots of $\widehat{J}$ and indicates that there is negligible difference in the sparsity of the estimates of the partial correlation matrix between model $NDPCSVS$ and model $DCPCSVS$. Both plots suggest that the partial correlations have an autoregressive type structure.



a) (b)

Figure 5.1: Image plots of $\widehat{C}$ for the pig growth rate data. Panel (a) is for model $NDPCSVS$. Panel (b) is for model $DCPCSVS$.



a) (b)

Figure 5.2: Image plots of $\widehat{J}$ for the pig growth rate data. Panel (a) is for model $NDPCSVS$. Panel (b) is for $DCPCSVS$.

5.4.2 Cow diet data

This data set consists of observations on 50 cows that are subjected to a diet additive. The data is cross-sectional and is described in Gelman et al. (2000,p.213-215) [2]. Each cow is assigned to one of four different levels of diet additive: 0% for the first 12 cows, 0.1% for cows 13-25, 0.2% for cows 26-38 and 0.3% for the remaining 12 cows. The following variables are also recorded for each cow, where 'p' indicates predictor, and 'r' indicates response:

p2 Lactation

p3 Age (mos)

[2]The data is available from http://www.stat.columbia.edu/∼gelman/book/data/.

p4 Initial weight (lb)

r1 Mean daily dry matter consumed (kg)

r2 Mean daily milk product (lb)

r3 Milk fat (%)

r4 Milk solids nonfat (%)

r5 Final weight (lb)

r6 Milk protein (%)

The first 3 variables were recorded before the additive was included in the diet and the last 6 variables were recorded after the additive was included in the diet. Following Cripps et al. (2005), we treat the 6 post-diet additive variables as the multivariate response and the diet additive and the 3 pre-diet additive variables as the predictors. We model the data as in (5.3) which allows the same covariates to have different regression coefficients for each element in the vector of the responses.

An interesting feature of this data is the relatively high correlation amongst some of the predictor variables. In particular, the correlation between lactation and age is 0.9624, the correlation between lactation and initial weight is 0.7504 and the correlation between age and initial weight is 0.7808.

The response vector for the tth cow is $y_t = (y_{t1}, y_{t2}, \ldots, y_{t6})'$ where y_t contains, in the following order, *Mean daily dry matter consumed, Mean daily milk product, Milk fat, Milk solids nonfat, Final weight* and *Milk protein*. The predictor vector for the ith cow is $x_t = (x_{t0}, x_{t1}, x_{t2}, x_{t3}, x_{t4})'$, where x_t contains, in the following order, an intercept, *Diet additive, Lactation, Age* and *Initial weight*. The matrix of regression coefficients in (5.3) for this example is,

$$B = \begin{bmatrix} \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \beta_{1,4} \\ \beta_{2,0} & \beta_{2,1} & \beta_{2,2} & \beta_{2,3} & \beta_{2,4} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \beta_{6,0} & \beta_{6,1} & \beta_{6,2} & \beta_{6,3} & \beta_{6,4} \end{bmatrix}.$$

We present analysis for both the τS_y and the equicorrelated forms of Φ because the results of Section 7.2 suggest that if Ω is sparse, then the equicorrelated prior will

Posterior mean

	$\beta_{1,0}$	$\beta_{1,1}$	$\beta_{1,2}$	$\beta_{1,3}$	$\beta_{1,4}$	$\beta_{2,0}$	$\beta_{2,1}$	$\beta_{2,2}$	$\beta_{2,3}$	$\beta_{2,4}$
$NDPCSVS$	9.5310	0.5236	0.0623	0.0010	0.0050	30.7891	-0.4925	0.2332	0.0084	0.0213
τS_y	10.2854	0.6567	0.0714	0.0017	0.0044	34.3323	-0.5668	0.3542	0.0103	0.0182
$equi$	9.0785	0.2399	0.0909	-0.0019	0.0055	26.5486	-0.0982	0.1089	0.0009	0.0251
	$\beta_{3,0}$	$\beta_{3,1}$	$\beta_{3,2}$	$\beta_{3,3}$	$\beta_{3,4}$	$\beta_{4,0}$	$\beta_{4,1}$	$\beta_{4,2}$	$\beta_{4,3}$	$\beta_{4,4}$
$NDPCSVS$	2.9592	2.0622	0.0316	0.0001	0.0001	8.5256	-0.0074	-0.0109	-0.0004	0.0000
τS_y	2.9647	2.0559	0.0294	0.0000	0.0001	8.5341	-0.0075	-0.0132	-0.0005	0.0000
$equi$	2.9616	2.1010	0.0291	0.0004	0.0001	8.5238	-0.0091	-0.0070	-0.0005	0.0000
	$\beta_{5,0}$	$\beta_{5,1}$	$\beta_{5,2}$	$\beta_{5,2}$	$\beta_{5,4}$	$\beta_{6,0}$	$\beta_{6,1}$	$\beta_{6,2}$	$\beta_{6,3}$	$\beta_{6,4}$
$NDPCSVS$	232.9853	-191.9362	0.3351	-0.0346	0.8080	3.2934	-0.0119	0.0009	0.0001	-0.0000
τS_y	238.9370	-200.4239	0.5443	-0.0393	0.8041	3.2968	-0.0138	0.0011	0.0001	-0.0000
$equi$	231.9985	-165.1042	0.0471	-0.0515	0.8066	3.2912	-0.0075	0.0008	0.0001	-0.0000

Table 5.2: Comparison of posterior means for the regression coefficients using model $NDPCSVS$ and $DCPCSVS$ for the τS_y and the equicorrelated forms of Φ, respectively, for the cow diet data.

Posterior sampling standard error

	$\beta_{1,0}$	$\beta_{1,1}$	$\beta_{1,2}$	$\beta_{1,3}$	$\beta_{1,4}$	$\beta_{2,0}$	$\beta_{2,1}$	$\beta_{2,2}$	$\beta_{2,3}$	$\beta_{2,4}$
$NDPCSVS$	3.3086	1.5375	0.2179	0.0119	0.0029	13.0697	3.1737	0.8675	0.0527	0.0116
τS_y	3.3432	0.5677	0.0862	0.0037	0.0016	11.5567	0.9113	0.4015	0.0170	0.0066
$equi$	2.8679	0.3685	0.1007	0.0059	0.0018	8.4345	0.7033	0.1580	0.0150	0.0082
	$\beta_{3,0}$	$\beta_{3,1}$	$\beta_{3,2}$	$\beta_{3,3}$	$\beta_{3,4}$	$\beta_{4,0}$	$\beta_{4,1}$	$\beta_{4,2}$	$\beta_{4,3}$	$\beta_{4,4}$
$NDPCSVS$	0.3669	0.5725	0.0688	0.0040	0.0003	0.1730	0.1014	0.0310	0.0019	0.0002
τS_y	0.9006	0.6456	0.0221	0.0012	0.0001	2.5598	0.0303	0.0109	0.0007	0.0001
$equi$	0.8959	0.6563	0.0220	0.0011	0.0001	2.5562	0.0313	0.0095	0.0006	0.0001
	$\beta_{5,0}$	$\beta_{5,1}$	$\beta_{5,2}$	$\beta_{5,2}$	$\beta_{5,4}$	$\beta_{6,0}$	$\beta_{6,1}$	$\beta_{6,2}$	$\beta_{6,3}$	$\beta_{6,4}$
$NDPCSVS$	82.8676	120.2146	4.5379	0.3559	0.0671	0.1259	0.0848	0.0103	0.0008	0.0001
τS_y	76.2745	72.2902	1.5802	0.1229	0.2428	0.9898	0.0274	0.0034	0.0002	0.000
$equi$	74.0973	64.4649	1.3500	0.1174	0.2433	0.9879	0.0276	0.0038	0.0003	0.0000

Table 5.3: Posterior standard errors for the regression coefficients using model $NDPCSVS$ and $DCPCSVS$ for the τA_y and equicorrelated forms of Φ, respectively, for the cow diet data.

work best, and if Ω is reasonably full, then the τS_y will work best, and we do not know, a priori, which case holds.

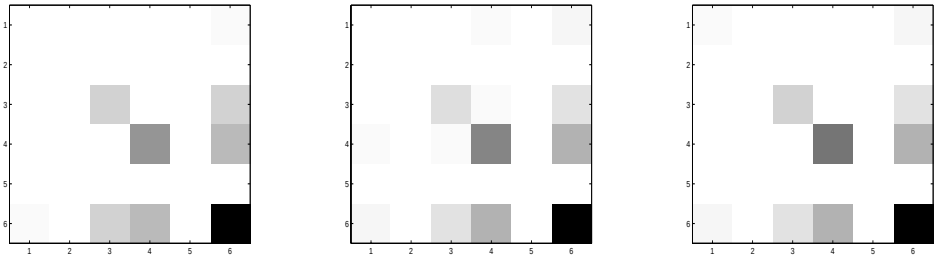
Tables 5.2, 5.3 and 5.4.2 compare the estimated means, standard errors and probabilities of being nonzero, respectively, of the regression coefficients, and shows that all models provide similar estimates.

Posterior probability of being non-zero.

	$\beta_{1,0}$	$\beta_{1,1}$	$\beta_{1,2}$	$\beta_{1,3}$	$\beta_{1,4}$	$\beta_{2,0}$	$\beta_{2,1}$	$\beta_{2,2}$	$\beta_{2,3}$	$\beta_{2,4}$
$NDPCSVS$	NA	0.1598	0.1453	0.1027	0.8208	NA	0.0816	0.1340	0.1116	0.8339
$DCPCSVS$ τS_y	NA	0.1819	0.1527	0.1131	0.7271	NA	0.0840	0.1676	0.1208	0.7349
$DCPCSVS$ equi.	NA	0.1013	0.1509	0.1043	0.8780	NA	0.0607	0.0910	0.0855	0.9379
	$\beta_{3,0}$	$\beta_{3,1}$	$\beta_{3,2}$	$\beta_{3,3}$	$\beta_{3,4}$	$\beta_{4,0}$	$\beta_{4,1}$	$\beta_{4,2}$	$\beta_{4,3}$	$\beta_{4,4}$
$NDPCSVS$	NA	0.9833	0.2918	0.1614	0.2203	NA	0.0663	0.1848	0.1436	0.1046
$DCPCSVS$ τS_y	NA	0.9778	0.2713	0.1464	0.2204	NA	0.0701	0.2092	0.1605	0.1100
$DCPCSVS$ *equi.*	NA	0.9919	0.2832	0.1650	0.2003	NA	0.0643	0.1348	0.1280	0.0932
	$\beta_{5,0}$	$\beta_{5,1}$	$\beta_{5,2}$	$\beta_{5,3}$	$\beta_{5,4}$	$\beta_{6,0}$	$\beta_{6,1}$	$\beta_{6,2}$	$\beta_{6,3}$	$\beta_{6,4}$
$NDPCSVS$	NA	0.7952	0.0717	0.0746	1.0000	NA	0.0780	0.0809	0.0819	0.1212
$DCPCSVS$ τS_y	NA	0.8057	0.0756	0.0756	1.0000	NA	0.0803	0.0845	0.0852	0.1306
$DCPCSVS$ *equi.*	NA	0.6848	0.0657	0.0759	1.0000	NA	0.0641	0.0703	0.0809	0.1116

Table 5.4: Posterior probabilities of the regression coefficients being nonzero for models $NDPCSVS$ and $DCPCSVS$ using the τS_y and equicorrelated forms of Φ, respectively, for the cow diet data. NA means not applicable as the coefficient is always included.

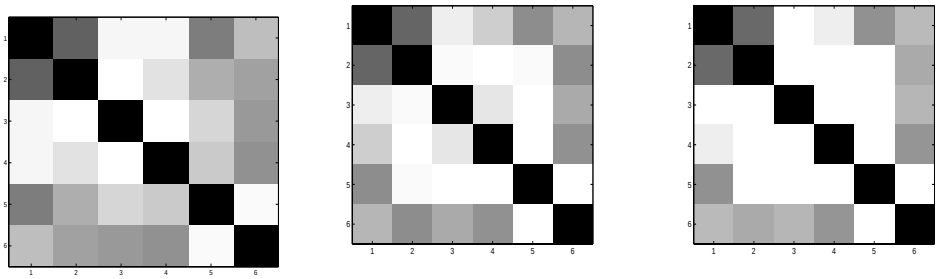
Figure 5.3 compares the image plots of $\widehat{\Omega}$ and suggests that there is little difference in the estimates of the models.



(a) (b) (c)

Figure 5.3: Image plots for the cow milk protein data. Panel (a) is the image plot of $\widehat{\Omega}$ as estimated by model $NDPCSVS$. Panel (b) is the same as estimated by $DCPCSVS$ using the τS_y form of Φ. Panel (c) is the same as estimated by model $DCPCSVS$ using the equicorrelated form of Φ.

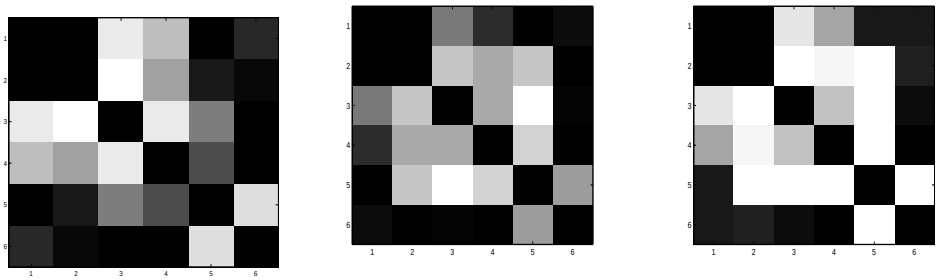
Figure 5.4 compares the image plots of $\widehat{C}$ and suggests that the estimates for $C_{1,U}$, and $C_{U,6}$ where $U = (2, \ldots, 6)$ are very similar in all three models. There are other regions of agreement in the estimates for C, but the τS_y $DCPCSVS$ estimate is closer to the $NDPCSVS$ estimate than the equicorrelated $DCPCSVS$ estimate.



(a) (b) (c)

Figure 5.4: Image plots for the cow milk protein data. Panel (a) is the image plot of $\widehat{C}$ as estimated by model $NDPCSVS$. Panel (b) is the same as estimated by model $DCPCSVS$ using the τS_y squares form of Φ. Panel (c) is the same as estimated by model $DCPCSVS$ using the equicorrelated form of Φ.

Figure 5.5 compares image plots of $\widehat{J}$, as estimated by models $NDPCSVS$, and $DCPCSVS$ using the τS_y and the equicorrelated forms of Φ, respectively. As expected, the equicorrelated prior estimate is the sparsest. The plots indicate that although there is some difference in the sparsity of the estimates, all models agree on a large proportion of edges.



(a) (b) (c)

Figure 5.5: Image plots for the cow milk protein data. Panel (a) is the image plot of $\widehat{J}$ for model $NDPCSVS$. Panel (b) is the same for model $DCPCSVS$ using the τS_y form of Φ. Panel (c) is the same for model $DCPCSVS$ using the equicorrelated form of Φ.

Figure 5.6 compares edge posterior probabilities. These are lowest for the equicorrelated $DCPCSVS$. Panel (a) shows that all three models are within 15% of each other on 6 of the 15 possible edges. Panel (c) shows that the posterior edge probability estimates of the τS_y $DCPCSVS$ model are within 30% of the $NDPCSVS$

estimates on every edge, and within 10% on 7 of the 15 edges. Panels (b) and (d) suggest that the τS_y $DCPCSVS$ estimates are closer to the $NDPCSVS$ estimates than the equicorrelated decomposable model, and that the τS_y $DCPCSVS$ estimates are closer to the $NDPCSVS$ estimates than to the equicorrelated $DCPCSVS$ estimates.

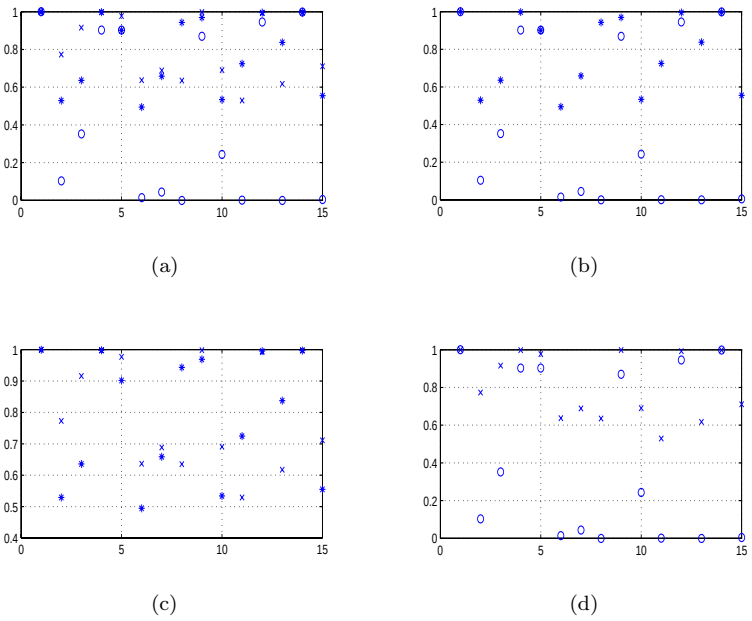


Figure 5.6: Edge posterior plots for the cow diet dataset. The edges are ordered in sequence along the x-axis as $(1,2),(1,3),\ldots,(1,6),(2,3),(2,4),\ldots,(5,6)$, and the y-axis is the corresponding posterior sampling probability of the edge being present. Panel (a) plots $NDPCSVS$ (*), $DCPCSVS$ using the equicorrelated form of Φ (o), and $DCPCSVS$ using the scaled sum of squares form of Φ (x). Panel (b) plots $NDPCSVS$ (*) compared with $DCPCSVS$ using the equicorrelated form of Φ (o). Panel (c) plots $NDPCSVS$ (*) compared with $DCPCSVS$ using the scaled sum of squares form of Φ (x). Panel (d) plots $DCPCSVS$ using the equicorrelated form of Φ (o) compared with $DCPCSVS$ using the scaled sum of squares form of Φ (x).

Figure 5.7 compares the graphs which consist of edges with posterior sampling probabilities of at least $75\%, 85\%, 95\%$ and 99% respectively. Panel (a) of each is the estimate of $NDPCSVS$, whilst (b) and (c) are those of $DCPCSVS$ using the τS_y and the equicorrelated forms of Φ, respectively.

Summary

For brevity, we call the graphs consisting of edges with at least $k\%$ posterior sampling probability the $k\%$ graphs. Note that the decomposable $k\%$ graphs need not be decomposable.

1. the two most likely edges for all models are the same: these are $e_1 = (1, 2)$ and $e_2 = (4, 6)$. This implies that *Mean daily milk product* and *Mean daily dry matter consumed* are completely dependent: i.e. cannot be made independent by conditioning on any subset of the remaining variables. Similarly, *Milk solids nonfat* and *Milk protein* are completely dependent.

2. (c) is a subgraph of both (a) and (b) in every case.

3. The DCP equicorrelated 75% and 85% graphs are the same, and equal to the τS_y 95% graph.

4. The equicorrelated 95% and 99% graphs are the same, and a subgraph of every graph.

5. The 95% graphs for NDP and DCP using the τS_y form of Φ are the same.

6. The NDP 99% graph is a subgraph of all the remaining models graphs, except the equicorrelated 99% graph.

7. At the 85% level, the graph in (d) is the graph in (f) plus the edges (2,5). The graph in (e) is the graph in (f) plus the edge (4,1).

8. At the 75% level, the graph in (a) is the graph in (c) plus the edges (5,4) and (5,2). It is not decomposable. The graph in (b) is the graph in (c) plus (5,6), (1,3) and (1,4).

We now give the details. We first consider point 7. above. Panels (d), (e) and (f) of Figure 5.7 are the 85% graphs for each model. Panel (d) is for model $NDPCSVS$ and Panels (e) and (f) are for model $DCPCSVS$ using the τS_y and the equicorrelated forms of Φ, respectively. The difference of the edge $e = (2, 5)$ means that model $NDPCSVS$ estimates that *r2=Mean daily milk product* is dependent on *r5=Final weight*, even after conditioning on all remaining variables. On the other

hand, because *r2=Mean daily milk product* is connected to, but not adjacent to *r5=Final weight* in Panels (e) and (f), model $DCPCSVS$ estimates that *Mean daily milk product* is independent of *Final weight*, after conditioning on all the rest. In the graphs of both $NDPCSVS$ and $DCPCSVS$, *r3=Milk fat* is made independent of *r4=Milk solids, nonfat*, by conditioning on *r6=Milk protein* because $r3 \overset{g}{\perp} r4|r6$. This means that *Milk solids nonfat* is irrelevant to knowing *Milk fat* in the situation where you know *Milk protein*: you only need to condition on *Milk protein* to make these two independent, and need not condition on, for example, *Mean daily dry matter consumed*, or any of the other remaining variables. On the other hand, the models disagree on the relation between *Mean daily milk product* and *Final weight*. Model $NDPCSVS$ says that these can never be made independent of one another, no matter what other variables you condition on. Model $DCPCSVS$ says that *Mean daily milk product* can be made independent of *Final weight* by conditioning on *Mean daily dry matter consumed*. This is arguably reasonable: if I made a guess about *Mean daily milk product*, then you told me *Final weight* had changed, then I would update my estimate accordingly. But if, before you told me the new *Final weight*, I gained information about the new *Mean daily dry matter consumed*, then I would get no further 'new' information by knowing *Final weight*. That is, *Final weight* is irrelevant to *Mean daily milk* in the situation where *Mean daily dry matter consumed* is known. This is reasonable because a cow's *Final weight* would be largely equivalent to the amount of daily dry matter consumed, given the *Initial weight* was the same. On the other hand, these variables cannot be made independent by knowing *Milk fat*. So *Final weight* is still relevant to *Mean daily milk* in the situation where only the *percentage* of *Milk fat* is known.

The 85% graphs of $DCPCSVS$ using the equicorrelated form of Φ and $DCPCSVS$ using the τS_y form of Φ differ in only one edge, $e = (1, 4)$.

We now consider the 99% graphs shown in Panels (k), (l) and (m) of Figure 5.7. The conclusion from model $NDPCSVS$ is that $\{v_3, v_4, v_6\} \perp\!\!\!\perp \{v_1, v_2, v_4\}$; i.e. that *Milk fat*, *Milk protein* and *Milk solids non fat*, are marginally independent of *Mean daily milk product*, *Final weight* and *Mean dry matter consumed*. On the other hand, the conclusion from model $NDPCSVS$ is that these are not marginally independent, but can be made independent by conditioning. In particular, there is a dependency between the *Mean daily milk product* and the *Milk protein* % that cannot be removed

by conditioning.

More inference about the conditional independencies between variables can be inferred from the graphs, but for brevity this discussion is omitted.

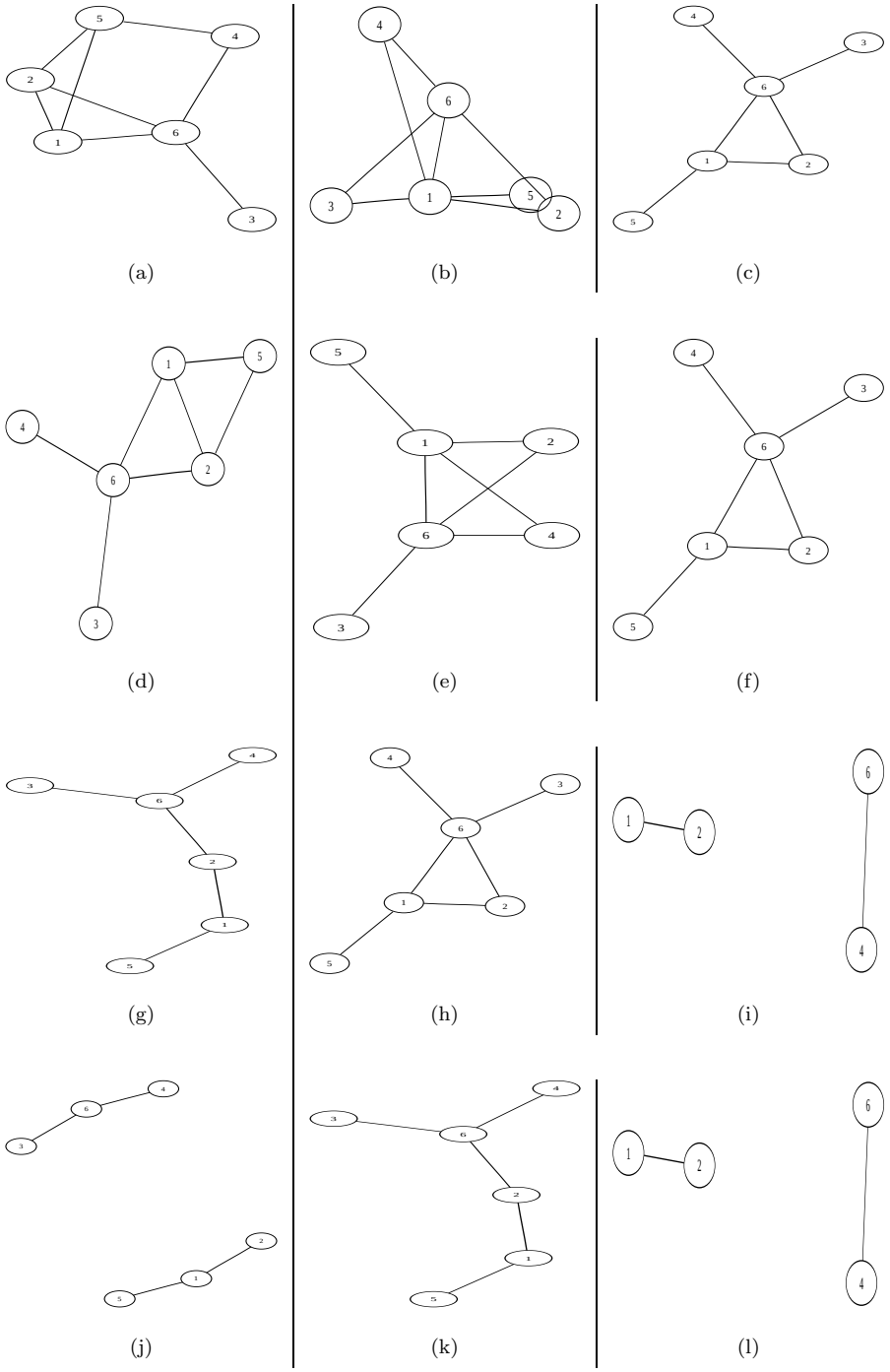
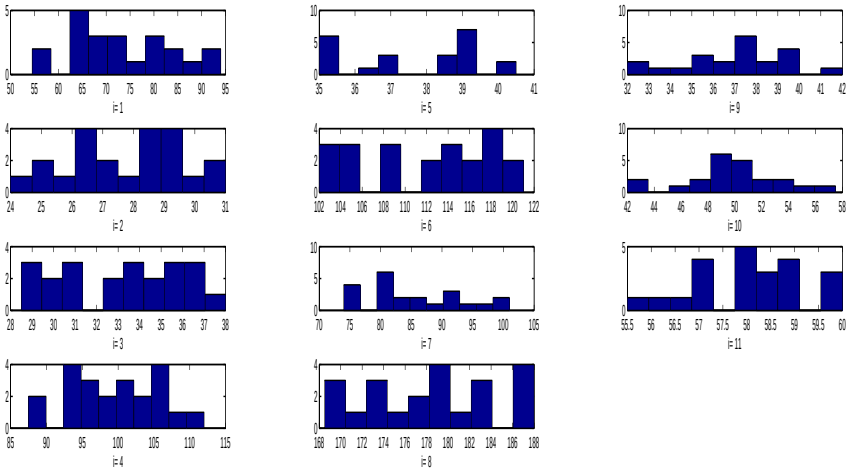


Figure 5.7: Graphs for the cow milk protein data. Panels (a), (b) and (c) are the 75% graphs for model $NDPCSVS$, and model $DCPCSVS$ using the scaled sum of squares and the equicorrelated forms of Φ, respectively. Panels (d), (e) and (f) are the same, at the 85% level. Panels (g), (h) and (i) are the same, at the 95% level. Panels (j), (k) and (l) are the same, at the 99% level.

5.4.3 Physical measurements data: model 1

In this section we compare the $NDPCSVS$ and $DCPCSVS$ models on a dataset consisting of the weight and various physical measurements for 22 male subjects aged 16 to 30. Subjects were randomly chosen volunteers, all in reasonably good health. Subjects were requested to slightly tense each muscle being measured to ensure measurement consistency. Apart from *Mass*, all measurements are in cm. (see Larner, M. (1996). Mass and its Relationship to Physical Measurements. MS305 Data Project, Department of Mathematics, University of Queensland.)[3] Figure 5.8 gives histogram plots of all eleven variables. For simplicity and because this is an exploratory analysis, we carry out the analysis as if the variables were multivariate Gaussian, though we note that for further analysis of the data we would investigate transformations of the variables to make this assumption more reasonable.



(a) (b) (c)

Figure 5.8: Histograms of the physical measurements dataset. Panel (a) is the histograms for the first to fourth variables. Panel (b) is the histograms for the fifth to eighth variables. Panel (c) is the histograms for the ninth to eleventh variables.

The dataset is interesting because it consists of only 22 observations on 11 variables, and there are reasonable obvious interrelationships between variables. Standard analysis would regress weight on the remaining variables. We instead use a

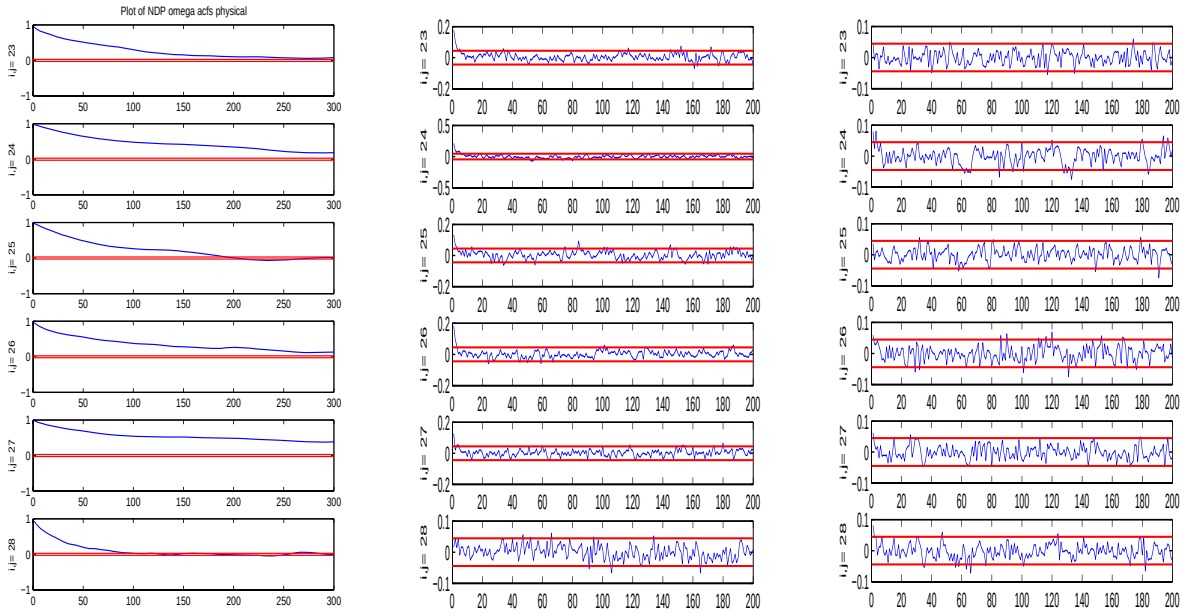
[3]The data can be downloaded from http://www.statsci.org/data/oz/physical.html.

graphical structure analysis. The graphical analysis allows us to identify (1) clusters of variables (i.e. cliques) from which we can infer variables that are redundant in the model, and, (2) variables which are adjacent to *Mass* in the graphical structure, and therefore cannot be made independent of *Mass* by conditioning on any subset of the remaining variables. These can be interpreted as the best predictors to include in the model for predicting the response variable *Mass*.

The $p = 11$ variables are indexed in the following order:

1. Mass: weight in kg,
2. Fore: maximum circumference of forearm,
3. Bicep: maximum circumference of bicep,
4. Chest: distance around chest directly under the armpits,
5. Neck: distance around neck, approximately halfway up,
6. Shoulders: distance around shoulders, measured around the peak of the shoulder blades
7. Waist: distance around waist, approximately trouser line,
8. Height: from top of head to toe,
9. Calf: maximum circumference of calf,
10. Thigh: circumference of thigh, measured halfway between the knee and the top of the leg,
11. Head: maximum circumference of head.

Figure 5.9 compares the autocorrelations for a representative selection of Ω_{ij} iterates. The autocorrelations decay rapidly to zero for the $DCPCSVS$ models, but not for the $NDPCSVS$ model. Note the scale on the y-axis in Figure 5.9: in Panel (a) the $NDPCSVS$ plot requires a y-axis going to 1.0 in all cases, but $DCPCSVS$ only needs 0.1-0.2 in general, and a maximum of 0.5. The difference in autocorrelation of the iterates between $NDPCSVS$ and $DCPCSVS$ is due to the greater efficiency of the sampling scheme in the decomposable case. Figure 5.9 also shows that the autocorrelations for the iterates of the τS_y form of Φ sampling scheme decay more rapidly to zero than the equicorrelated sampling scheme.



(a) (b) (c)

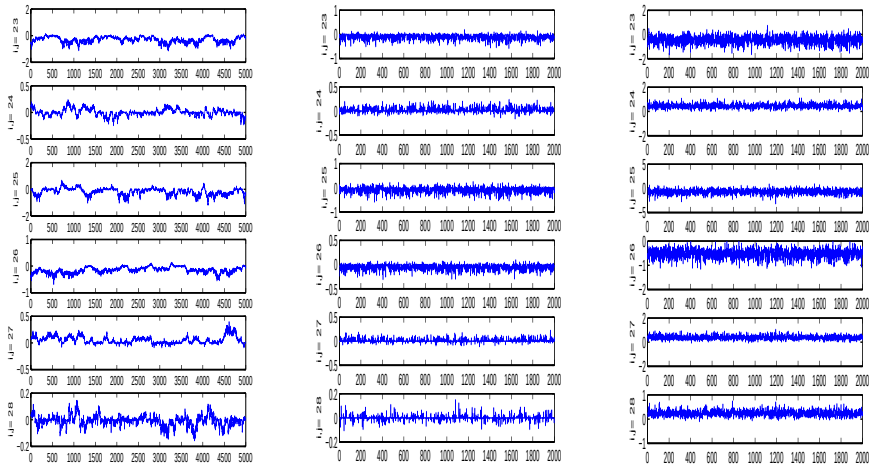
Figure 5.9: Autocorrelations in the iterates of the Ω_{ij} for a representative selection of Ω_{ij} and the physical measurements dataset. Panel (a) plots the autocorrelations for $NDPCSVS$. Panel (b) is the autocorrelations for $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is the autocorrelations for $DCPCSVS$ using the τS_y form of Φ.

Figure 5.10 shows that all models have converged. In particular, it shows that the high autocorrelations for $NDPCSVS$ are not because the sampler is not in the stationary distribution.

Figure 5.11 shows that the sampling average of Ω is similar for each model. There are definite regions of agreement between all three models: for example, the very sparse area in the right hand bottom corner, and that the second variable, *Maximum circumference of forearm*, shows partial correlation with the greatest number of the remaining variables.

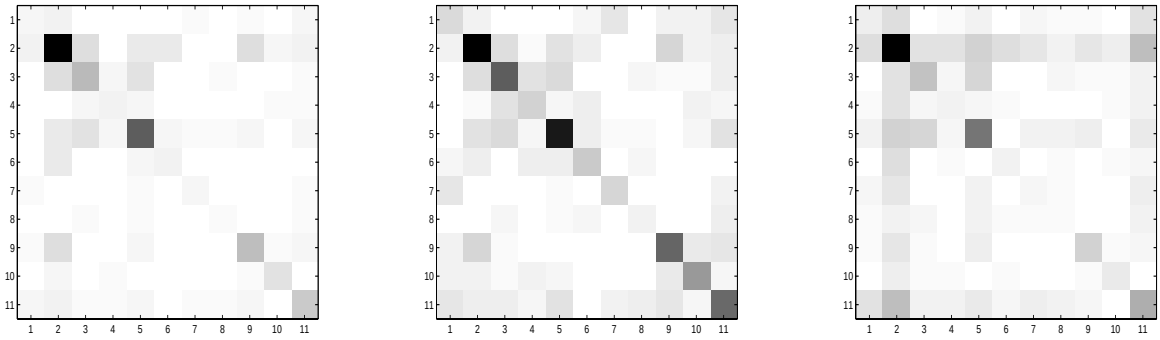
Figure 5.12 are image plots of the sampling averages of C for all 3 models, and suggests that all 3 models agree on the relatively strong relationship between $y_1 = Mass$ and $y_7 = Waist$. The image plots for $NDPCSVS$ and equicorrelated $DCPCSVS$ posterior means of $C_{U,U}$, where $y_U' = (y_1, \ldots, y_7)$, appear almost identical.

Figure 5.13 compares the image plots of $\widehat{J}$ and supports similar conclusions to those reported for Figure 5.12.



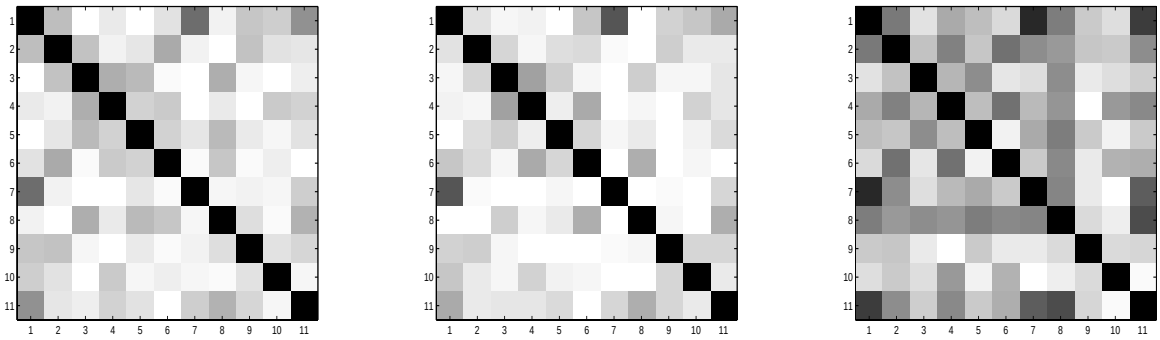
(a) (b) (c)

Figure 5.10: Iterates of Ω_{ij} for a the same representative selection of indicies shown in Figure 5.9. Panel (a) is the iterates for $NDPCSVS$. Panel (b) is the same for $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is the same for $DCPCSVS$ using the scaled sum of squares form of Φ.



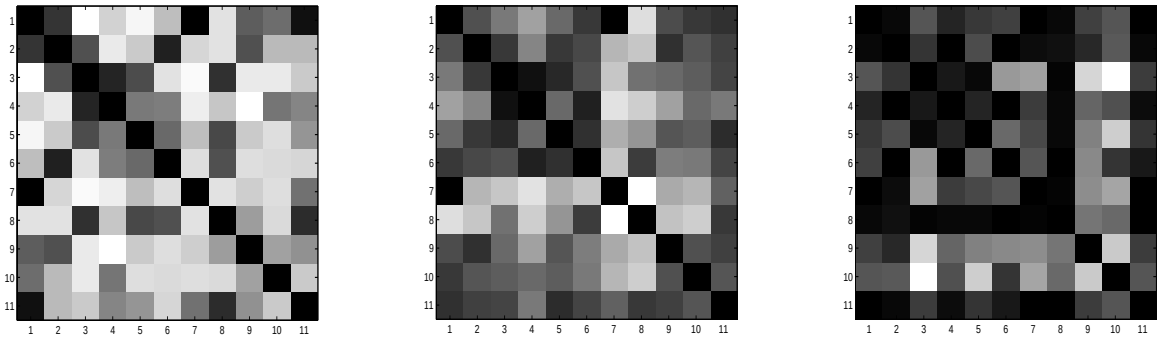
(a) (b) (c)

Figure 5.11: Image plots of $\widehat{\Omega}$ for the physical measurements dataset. Panel (a) is for $NDPCSVS$. Panel (b) is for $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is for $DCPCSVS$ using the τS_y form of Φ.



(a) (b) (c)

Figure 5.12: Image plots of $\widehat{C}$ for the physical measurements dataset. Panel (a) is for $NDPCSVS$. Panel (b) is forthe $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is for $DCPCSVS$ using the τS_y form of Φ.



(a) (b) (c)

Figure 5.13: Image plots of $\widehat{J}$. Panel (a) is for $NDPCSVS$. Panel (b) is for $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is for $DCPCSVS$ using the τS_y form of Φ.

Figure 5.14 compares edge posterior probabilities. In general, tbe τS_y $DCPCSVS$ model gives the highest posterior probabilities, and the equicorrelated $DCPCSVS$ model gives the lowest. Panel (a) shows that all three models are within 15% of each other on 10 of the 55 possible edges. Panel (c) shows that the τS_y $DCPCSVS$ posterior edge probability estimates are within 30% of the $NDPCSVS$ estimates on every edge, and within 10% on 12 of the 55 edges. Panels (b) and (d) suggest that the τS_y $DCPCSVS$ estimates are closer to the $NDPCSVS$ estimates than the equicorrelated decomposable model estimates, and that the τS_y $DCPCSVS$ estimates are closer to the $NDPCSVS$ estimates than to the equicorrelated $DCPCSVS$ estimates.

Figure 5.15 shows the $70\%, 90\%$ and 95% graphs, respectively. At the 50% level, all graphs are full and these are omitted for brevity. The equicorrelated decomposable prior consistently gives the sparsest estimates, followed by the $NDPCSVS$ and the τS_y decomposable model estimates, respectively. The graphs show that $y_1 = Mass$ and $y_2 = Waist$ are dependent in all cases, and there is no conditioning set of variables that can make them independent. This conclusion makes reasonable sense. Interestingly, the equicorrelated $DCPCSVS$ model and the $NDPCSVS$ models imply that $Mass$ is conditionally dependent on either $y_{10} = Thigh$ or $y_{11} = Head$, respectively. The sample means of y_{10} and y_{11} are more similar to each other than the means of any of the remaining variables, so these models can be interpreted as giving very similar inference about the most likely edges.

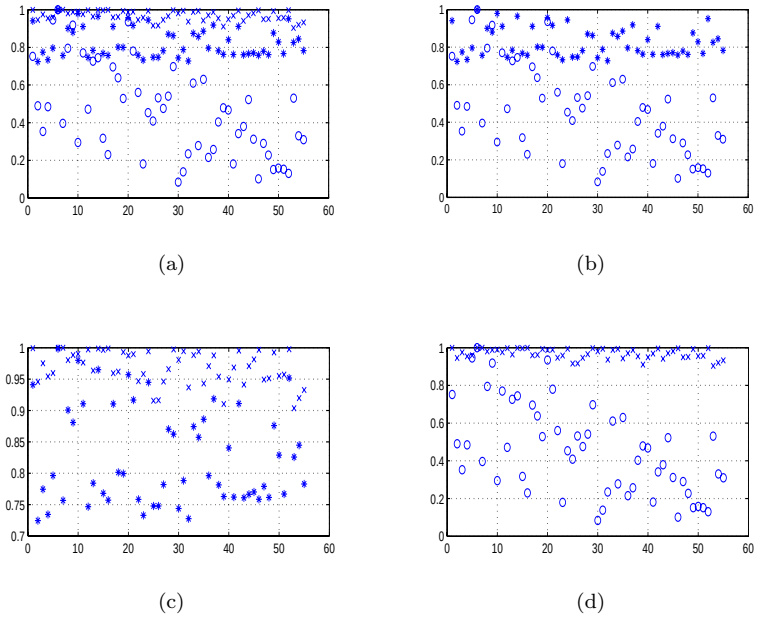


Figure 5.14: Edge posterior plots for the physical measurements dataset. The edges are ordered in sequence along the x-axis as $(1,2),(1,3),\ldots,(1,11),(2,3),(2,4),\ldots,(10,11)$, and the y-axis is the corresponding posterior sampling probability of the edge being present. Panel (a) plots $NDPCSVS$ (*), $DCPCSVS$ using the equicorrelated form of Φ (o), and $DCPCSVS$ using the τS_y form of Φ (x). Panel (b) plots $NDPCSVS$ (*) compared with $DCPCSVS$ using the equicorrelated form of Φ (o). Panel (c) plots $NDPCSVS$ (*) compared with $DCPCSVS$ using the scaled sum of squares form of Φ (x). Panel (d) plots $DCPCSVS$ using the equicorrelated form of Φ (o) compared with $DCPCSVS$ using the scaled sum of squares form of Φ (x).

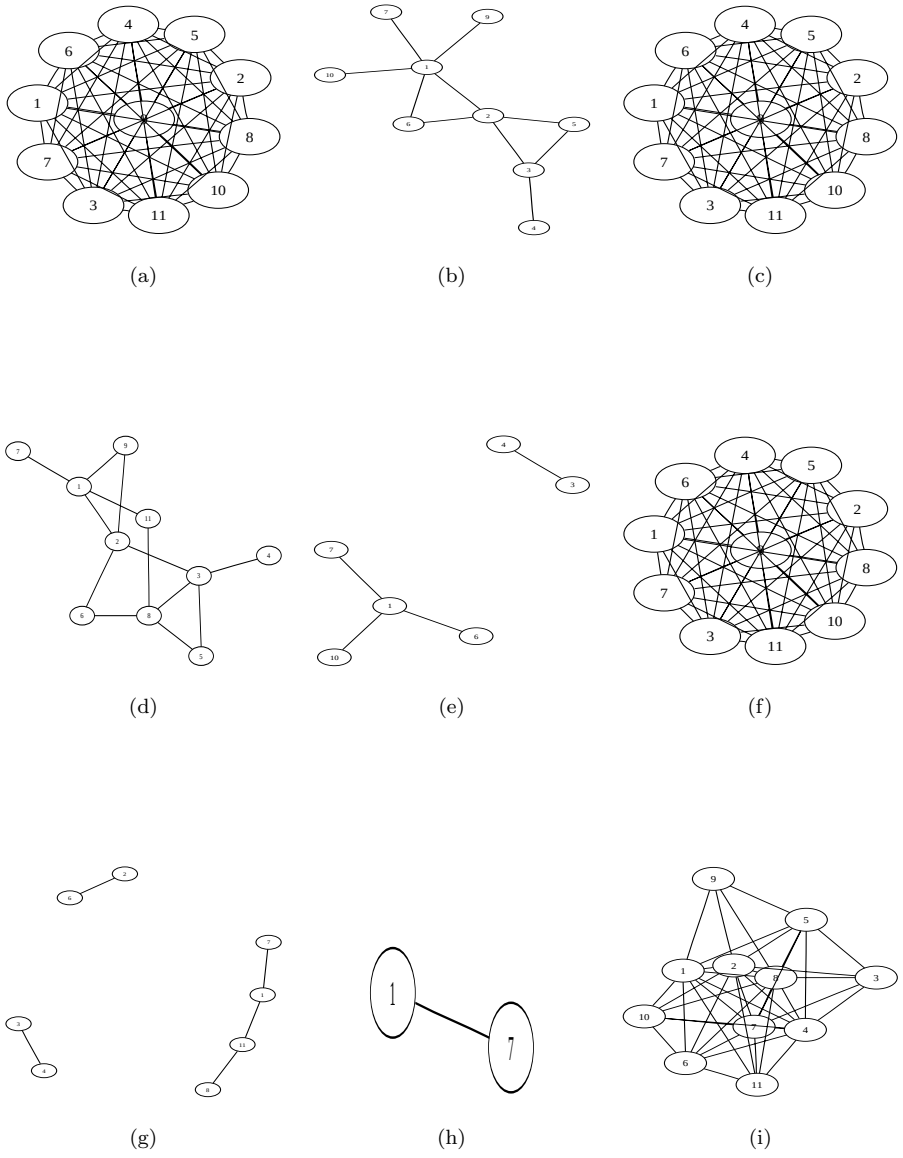


Figure 5.15: Graph pictures for the physical measurements dataset. Panels (a), (d) and (g) are $70\%, 90\%$ and 95% graphs, respectively, for model $NDPCSVS$. Panels (b), (e) and (h) are the same, for model $DCPCSVS$ using the equicorrelated form of Φ. Panels (c), (f) and (i) are the same, for by model $DCPCSVS$ using the τS_y form of Φ.

The summary analysis from Larner, M. (1996) is given below. It is the output of *leaps* which does variable selection based on minimising the Cp Mallows criteria score. It does not communicate the independence interrelations of the graphical analysis just given. (See further discussion in Section 6.5.)

```
> leaps.mass <- leaps(physical[,2:11],Mass,nbest=3)
> df.mass <- data.frame(p=leaps.mass$size,Cp=leaps.mass$Cp)
> round(df.mass,2)
                                                          p     Cp
                                                  Waist   2  60.50
                                                   Fore   2  74.80
                                               Shoulder   2 110.36
                                             Fore,Waist   3  14.70
                                             Waist,Calf   3  25.25
                                          Shoulder,Waist   3  29.54
                                       Fore,Waist,Height   4   7.45
                                         Fore,Waist,Calf   4  11.18
                                        Fore,Waist,Thigh   4  12.21
                                  Fore,Waist,Height,Thigh   5   4.44
                                   Fore,Waist,Height,Calf   5   6.10
                                   Fore,Waist,Height,Head   5   6.83
                             Fore,Waist,Height,Thigh,Head   6   4.14
                             Fore,Waist,Height,Calf,Thigh   6   4.82
                              Fore,Waist,Height,Calf,Head   6   5.35
                        Fore,Waist,Height,Calf,Thigh,Head   7   4.38
                        Fore,Chest,Waist,Height,Calf,Head   7   4.81
                       Fore,Chest,Waist,Height,Thigh,Head   7   5.50
                 Fore,Chest,Waist,Height,Calf,Thigh,Head   8   5.47
                 Fore,Bicep,Waist,Height,Calf,Thigh,Head   8   6.07
             Fore,Shoulder,Waist,Height,Calf,Thigh,Head   8   6.12
            Fore,Chest,Neck,Waist,Height,Calf,Thigh,Head   9   7.13
        Fore,Chest,Shoulder,Waist,Height,Calf,Thigh,Head   9   7.45
           Fore,Bicep,Chest,Waist,Height,Calf,Thigh,Head   9   7.47
      Fore,Bicep,Chest,Neck,Waist,Height,Calf,Thigh,Head  10   9.01
   Fore,Chest,Neck,Shoulder,Waist,Height,Calf,Thigh,Head  10   9.10
  Fore,Bicep,Chest,Shoulder,Waist,Height,Calf,Thigh,Head  10   9.45
Fore,Bicep,Chest,Neck,Shoulder,Waist,Height,Calf,Thigh,Head 11  11.00

> lm.mass <-lm(Mass~Fore+Waist+Height+Thigh)
> summary(lm.mass,cor=F)

Call: lm(formula = Mass ~ Fore + Waist + Height + Thigh)
Residuals:
    Min      1Q  Median     3Q   Max
 -3.882 -0.6756 -0.1017 0.9641 4.992

Coefficients:
               Value Std. Error   t value  Pr(>|t|)
(Intercept) -113.3120   14.6391    -7.7404    0.0000
```

```
      Fore    2.0356    0.4624    4.4020    0.0004
     Waist    0.6469    0.1043    6.2015    0.0000
    Height    0.2717    0.0855    3.1789    0.0055
     Thigh    0.5401    0.2374    2.2750    0.0361

Residual standard error: 2.249 on 17 degrees of freedom
Multiple R-Squared: 0.9659
F-statistic: 120.5 on 4 and 17 degrees of freedom, the p-value is 3.079e-012
```

5.5 HIV data analysis

In this section we use the $DCPCSVS$ model on a confidential dataset with the kind permission of the National Centre for HIV research in Sydney. This data set consists of $n = 14$ observations on a single patient at t weeks either on or off treatment. The $p = 24$ variables measured are cell counts of different phenotypes within each of the $CD4+$ and $CD8+$ T cells. Based on the results in Section 7.2, we use the equicorrelated form of Φ and the size-based prior for the decomposable graphs.

Let $y = (y_{t1}, \ldots, y_{tp})'$ be the observed cell counts at week t. For each $j = 1, \ldots, p$, let y_{tj} be the cell count of phenotype j, measured t weeks from the beginning of a $z_t = 0$ (off) or $z_t = 1$ (on) treatment period. We assume the piecewise linear model

$$y_{tj} = \beta_0^j + \beta_1^j t + \beta_2^j z_t + \epsilon_{t,j} \text{ for } j = 1, \ldots, p \tag{5.11}$$

where $z_t = 1$ if the patient is treated at time t, and $z_t = 0$ otherwise.

Writing (5.11) in the notation of (5.1),

$$y_t = x_t\beta + e_t, \quad e_t \sim N(0, \Sigma), \tag{5.12}$$

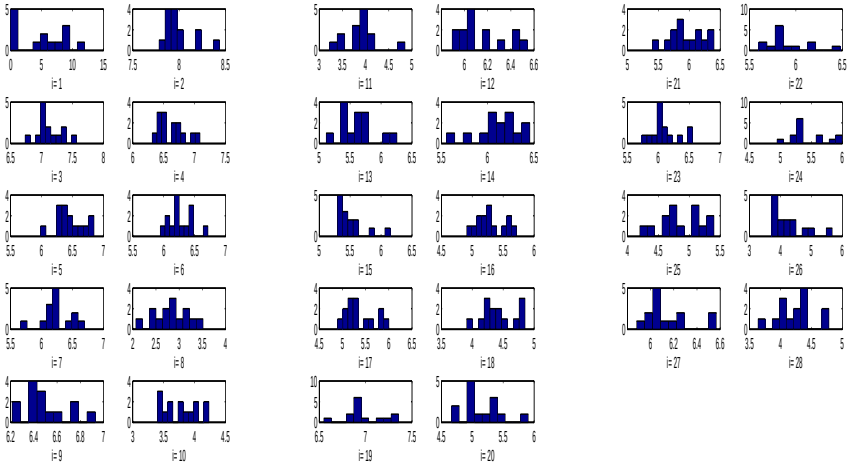
where $\Sigma \sim HIW(g, \delta, \Phi)$ the matrix x_t of covariates is of dimension $p \times 3p$ and is

$$x_t = \begin{bmatrix} 1 & t & z_t & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 1 & t & z_t & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & t & z_t & 0 & 0 & 0 & \ldots \\ \ldots & \ldots & & & & & & & & & & & \ldots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 1 & t & z_t \end{bmatrix}$$

; β is the corresponding $3p \times 1$ vector of regression coefficients consisting of $p/3$ subvectors $\beta^j = (\beta_0^j, \beta_1^j, \beta_2^j)'$ for each $j = 1, \ldots, p$.

Figure 5.16 shows the histograms of the natural logarithm of the variable values for $y_1, \ldots, y_{28}$ for the original dataset which includes variables which are knonw linear combinations of the remaining variables. For simplicity and because this is an exploratory

analysis, we take the logarithm of the entire dataset and assume a multivariate normal model, though we note that for further analysis of the data for biomedical inference we would investigate other transformations of some of the variables.



(a) (b) (c)

Figure 5.16: Histograms of the natural logarithm of the full HIV dataset. Panel (a) is the histograms for $\log(y_1), \ldots, \log(y_{10})$. Panel (b) is the histograms for $\log(y_{11}), \ldots, \log(y_{20})$. Panel (c) is the histograms for $\log(y_{21}), \ldots, \log(y_{28})$.

Let $V = \{1, \ldots, 24\}$ be the index set of the 24 selected variables of interest. In order to ensure non-singularity of the sample covariance, and because we only have $n = 14$ observations on $p = 24$ variables, we first analyse subsets of variables of sizes $p = 6, \ldots, 13 < n$ such that the variables within each subset are known a priori to be linearly independent, ensuring that the sample covariance matrix is non-singular.

The first empirical result is that the estimates on the intersections of such subsets of variables are the same. Panels (a) and (b), respectively, of Figure 5.17 shows the 75% graphs for 2 typical index subsets: $V3 = \{1, 4, 5, 6, 7, 8, 13, 14, 15, 16, 20, 23, 24\}$ and $V4 = \{5, 6, 7, 8, 11, 13, 14, 15, 16, 19, 20, 23, 24\}$. Consider $A = \{5, 14, 23, 13\}$, $B = \{6, 8\}$ and $C = \{16, 15, 20\}$, which are subsets of both $V3$ and $V4$. Figure 5.17 shows that the estimates of the induced subgraphs on the subsets A, B and C are the same in both cases, after allowing for disconnectedness about variables which are not common to both $V3$ and $V4$. For example, the subset $V3$ does not include the connecting variables y_4 and y_{11}, and so the induced subgraphs on A, B and C, respectively, are disconnected in Panel (a) but

connected in Panel (b). For subsets with non empty intersection, where the corresponding sample covariance was positive definite, this empirical result was typical. For brevity we report this example only.

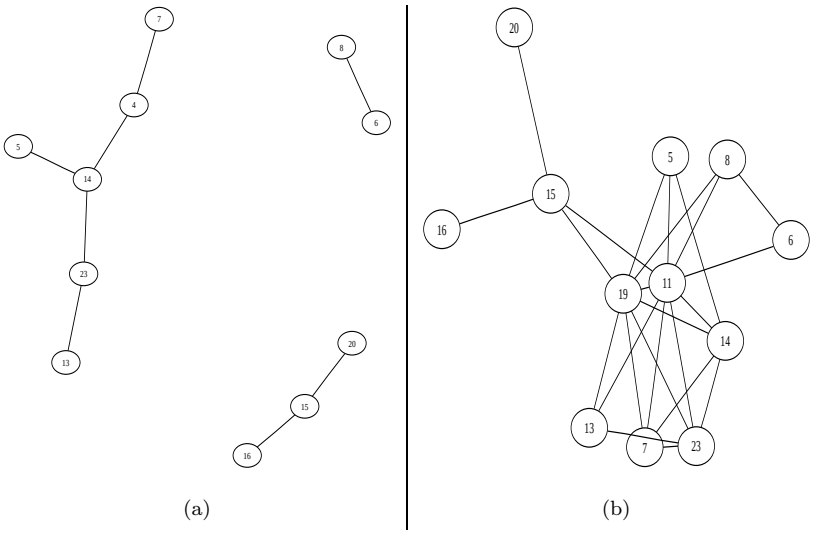


Figure 5.17: The 75% graphs for subsets of variables $V3$ and $V4$ with nonempty intersection. In this case, the associated covariances are known a priori to be positive definite. Panel (a) is for the index set $V3$. Panel (b) is for the index set $V4$. The induced subgraphs on common subsets of $V3$ and $V4$ are the same in each panel.

The second empirical result is that the first empirical result just discussed remains true when the condition for positive definiteness, that $p < n$, is removed. Consider the typical index subset $V2 = \{1, 4, 5, 6, 7, 8, 10, 12, 14, 15, 16, 17, 19, 21, 22, 23\}$ which has nonempty intersection with $V3$, but for which $V2 \backslash V3 \neq \emptyset$. It is known a priori that there are no linear relationships between the variables in y_{V2}, but because $|V2| = 16 > n = 14$ we know that the sample covariance is not positive definite. The graph in Panel (b) of Figure 5.18 is the 75% graph. We present the estimate for $V3$ again in Panel (a) to facilitate comparison and discussion. Recall that the eigenvalues of the sample covariance for $V3$ are significantly greater than zero. Panels (a) and (b) show that the model estimates agree on the induced subgraphs on the common vertices.

The third empirical result is that the results just discussed still hold when we take unions of index subsets such that $p > n$ and the sample covariance has many eigenvalues that are not significantly greater than zero. That is, even in this case the induced subgraphs on the intersections are the same. The graph in Panel (c) of Figure 5.18 is the estimate for the union of the two subsets $V2$ and $V3$. Panel (c) shows that the induced subgraphs g_{V3}

and g_{V2} are the same for both model estimates, and agree with the estimates in Panels (a) and (b), respectively.

In order to test this robustness of the model to the subsets chosen, we next consider the index subset $V8 = V2 \cup V3 \cup \{18\}$, the largest subset for which it is known that there are no linear relationships between the variables, but for which the sample covariance is singular because $p = 20 > n$.[4] Panel (d) of Figure 5.18 is the graph 75% graph. In Panel (d), the induced subgraphs of $V3$ and $V2$ are not exactly the same the graphs in Panels (a) and (b). However, when we allow for the exponentially larger sampling space by considering the 65%graphs, we see that all induced subgraphs agree. The graphs in Panels (e) and (f) are for the same variables as Panels (c) and (d), respectively, but consist of edges with posterior sampling probability at least 65%. Panels (e) and (f) more obviously contain as induced subgraphs the graphs in Panels (a), (b) and (c). It arguably better to consider the induced subgraphs of (e) and (f) rather than of (c) and (d), because the superexponential increase in the number of possible edges from $p = 13$ to $p = 20$ implies a lower expected posterior probability for every edge.

The fourth empirical result is that the third empirical result holds even when we know, a priori, that there are linear relationships between the variables because of the way the variable values are computed. We find empirically that we can still use graphical methods to obtain well behaved positive definite estimates of Ω and Σ.

Figure 5.19 compares the image plot of the inverse sample covariance to image plots of different $DCPCSVS$ model matrix estimates for $V = \{1, \ldots, 24\}$. In this case, it is known a priori that linear relationships exist between the variables because of the way their values are computed. Panel (a) is the inverse of the sample covariance. Because the eigenvalues are not precisely zero, MATLAB allows us to calculate this inverse. Panel (b) is the inverse of the average of the iterates of Σ for model $DCPCSVS$ which is positive definite by construction. Panel (c) is the average of the iterates of Ω for model $DCPCSVS$. Panel (d) is the average of the iterates of C for model $DCPCSVS$.

The matrix estimates do not allow us to interpret easily the conditional independencies. In particular, it is difficult to discern the vertices which are connected but not adjacent and the sets S that separate variables of particular interest. Figure 5.20 are the 65%, 70%, 75%, 80%, 85%, 90% and 95% graphs, respectively. The relationships that we discovered are of interest to the biomedical researchers who supplied the data because we can formulate

[4] In actual fact, 22 of the total 24 variables are theoretically not linearly related, however in this case there will be two subsets of variables whose total values are almost equal. We therefore chose to exclude these two variables also.

relationships between the variables in a similar way to those formulated in the analysis of the cow diet dataset in Section 5.4.2. In particular, we can postulate which variables are irrelevant to knowing key variables of interest after conditioning on other variables, and which variables are always dependent no matter which other variables are known.

The final empirical result is that the β coefficient estimates are analogously consistent. That is, regardles of the subset of variables chosen on which to run the $DCPCSVS$ model, the estimates β_2^j for $j = 1, 21, 24$ were at least 10% in every case. All the remaining β_2^j and all the β_1^j are less than 10%. This result remained true when we ran the model on the whole $p = 24$ variables in which there are known linear relationships.

As a check on the sensitivity of the inference to some of the assumptions in the model, we compared the graphical estimates for $V3$ under the following 4 alternatives: (1) the dataset consisting of the natural logarithm of y_1 only, and the remaining data left in its original form; (2) the dataset consisting of $\log(\log(y_1))$ and $\log(y_{V3\setminus 1})$; and (3) the natural logarithm of all variables, but for a model in which there are no covariates. Figure 5.21 shows that the estimates of the graphical structures are surprisingly similar for each of these modeling assumptions.

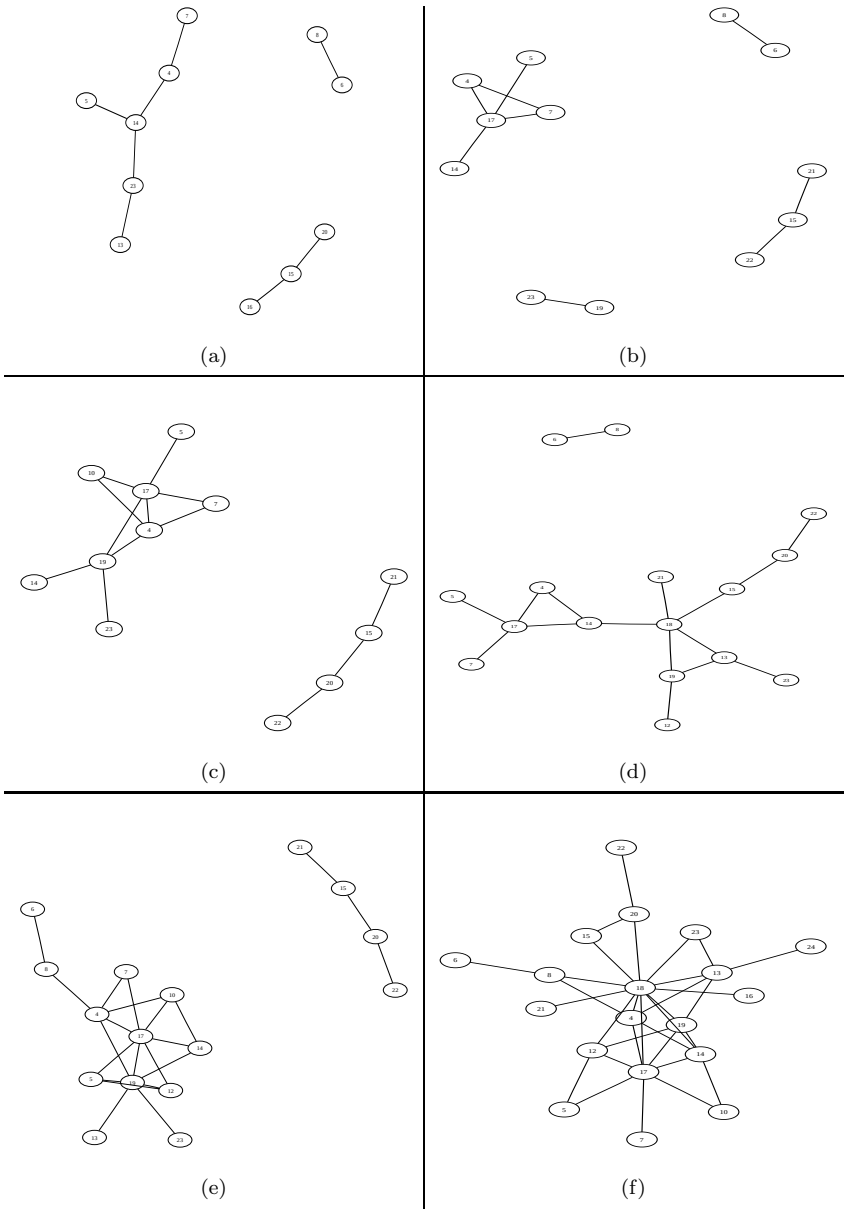


Figure 5.18: The 75% graphs for different subsets of variables from the HIV dataset. Panel (a) is the graph for $V3$, where $p = |V3| = 13 < n$ and the sample covariance is positive definite with eigenvalues significantly larger than zero. Panel (b) is the graph for $V2$, which has nonempty intersection with $V3$ but does not properly contain $V3$. In this case, $p = |V2| = 16 > n$ so the sample covariance is singular, but we know a priori that there are no linear relationships between the variables. Panel (c) is the graph for $V7 = V2 \cup V3$. In this case, $p = |V7| = 19 > n$ so the sample covariance is singular, but we know that there are no linear relationships between the variables. Panel (d) is the graph for $V8 = v2 \cup v3 \cup \{18\}$. This is the largest possible subset in which we know that there are no linear relationships and for which $p > n$. Panels (e) and (f) are the same as Panels (d) and (e), but are for the 65% graphs.

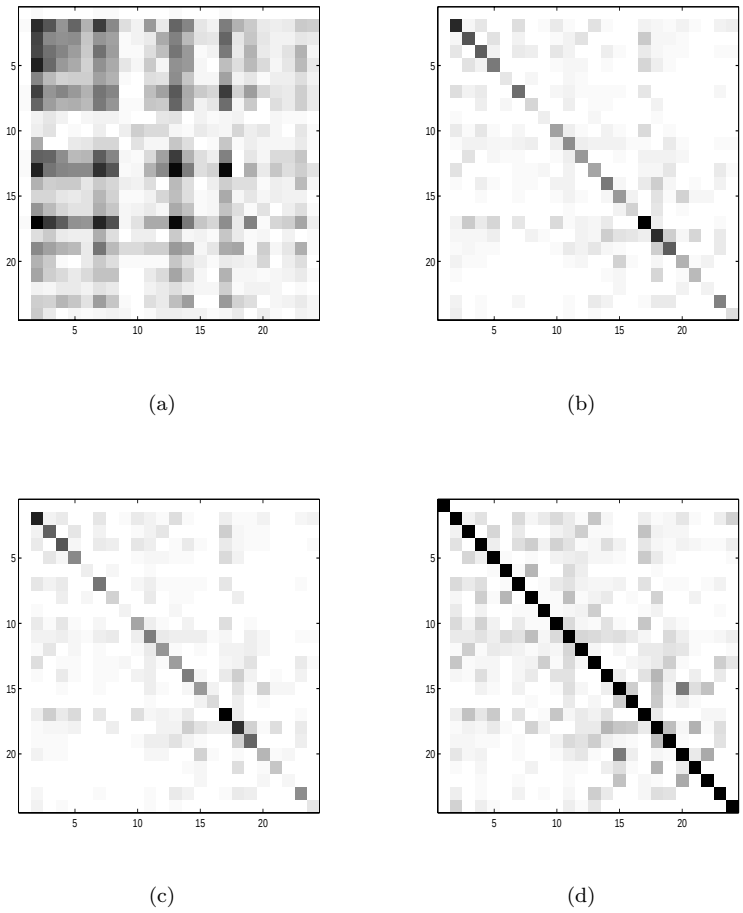


Figure 5.19: Image plots of various matrix estimates for the HIV dataset. Panel (a) is the inverse of the sample covariance. Panel (b) is the inverse of the average of the iterates of Σ for the $DCPCSVS$ model. Panel (c) is the average of the iterates of Ω for the $DCPCSVS$ model. Panel (d) is the average of the iterates of C for the $DCPCSVS$ model.

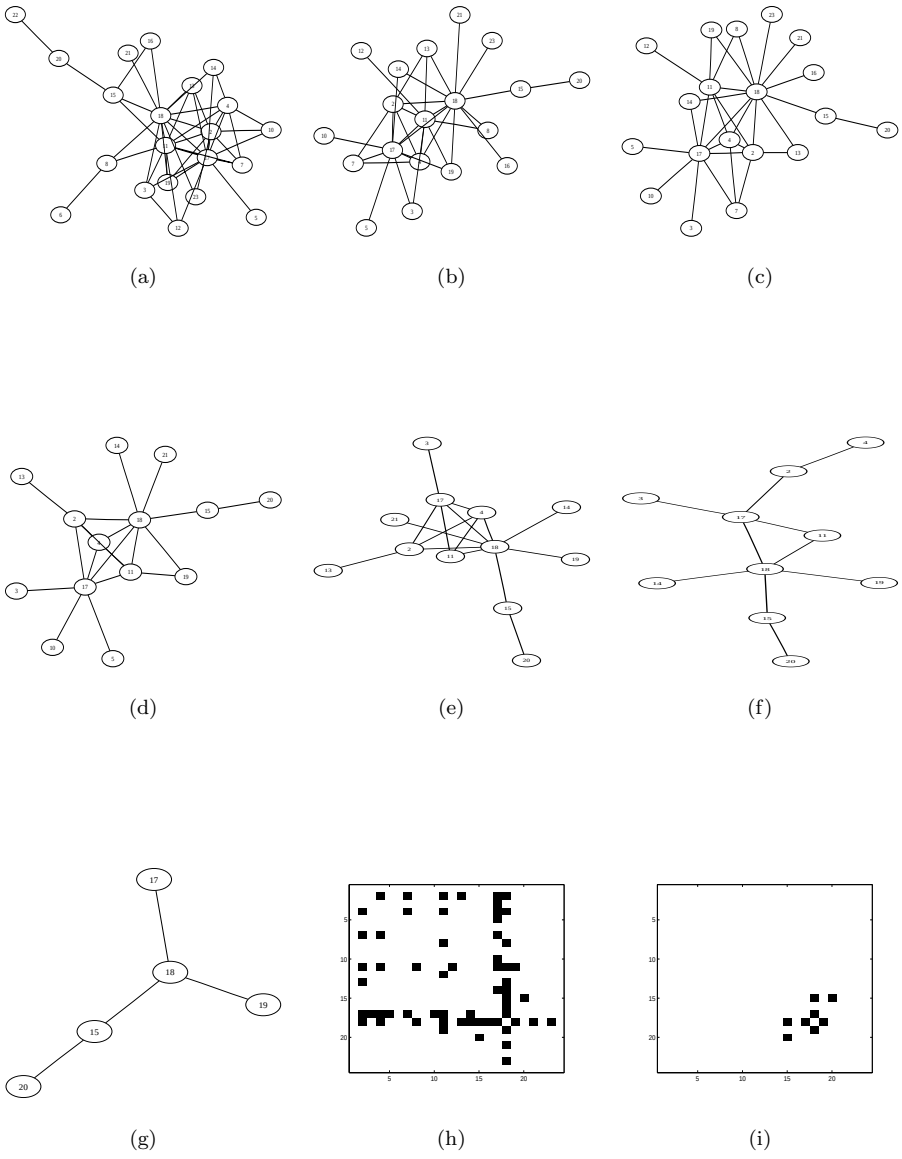


Figure 5.20: Graphs for the HIV dataset. Panels (a) to (g) are the 65%, 70%, 75%, 80%, 85%, 90% and 95% graphs, respectively. Panels (h) and (i) are the image plots of the adjacency matrices for the 75% and 95% graphs, respectively.

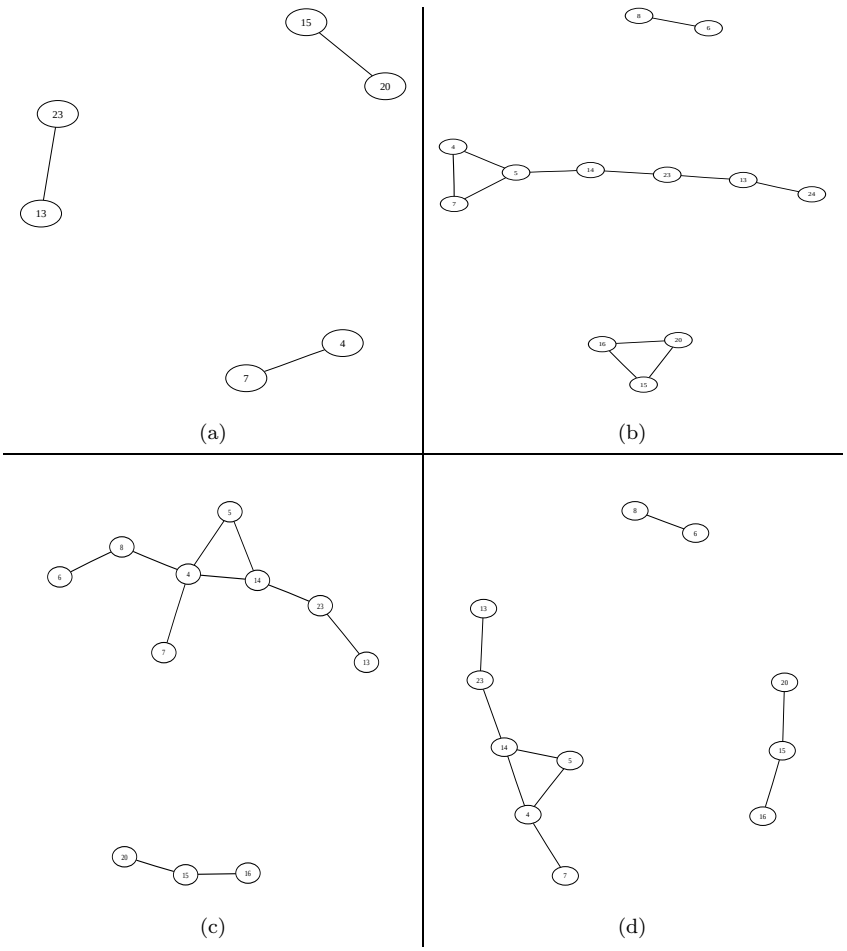


Figure 5.21: Graphs of different transformations for the subset $V3$ from the HIV dataset. Panel (a) is the 75% graph for the variables y_{V3} but taking $\log(y_1)$ only, and $y_{(V3\setminus 1)}$, where log indicates the natural logarithm. Panel (b) is the same as Panel (a), but for edges with posterior sampling probability at least 50%. Panel (c) is the same as Panel (a), but for $\log(\log(y_1))$ and $\log(y_{V3\setminus 1})$. Panel (d) is the same as Panel (a) in figure 5.18, but the covariates x_t are omitted from the model.

Chapter 6

Reduced conditional sampling for variable and covariance selection in multivariate regression models

6.1 Introduction

This chapter presents a new sampling scheme for variable and covariance selection in multivariate regression models which generates the variable and edge indicators with both the regression coefficients and the covariance matrix integrated out. This means that the new sampling scheme is potentially more efficient than that given in Chapter 5, and hence it is also a more efficient sampling scheme than the one used by Cripps et al. (2005).

The chapter is organised as follows. Section 6.2 describes the model, the priors and gives an expression for the marginal likelihood with both the vector of coefficients and the covariance matrix integrated out. Section 6.3 describes the sampling scheme used in this chapter and explains the difference between it and the sampling scheme introduced in Chapter 5. Section 6.4 compares the performance of the sampling scheme introduced in this chapter to the sampling scheme introduced in Chapter 5 and the sampler used in Cripps et al. (2005). The comparisons are made by applying all three schemes to the cow diet and physical measurements data sets introduced in Chapter 5. To illustrate that graphical analysis gives a richer interpretation of the data than regression analysis, two different regression models are used to analyse the physical measurements data.

6.2 Model description

6.2.1 Introduction

For $t = 1, \ldots, n$, consider the multivariate regression model

$$y_t = Bx_t + e_t, \quad e_t \sim N(0, \Sigma), \quad \Sigma \sim HIW(g, \delta, \Phi), \tag{6.1}$$

where y_t is $p \times 1$, B is a $p \times q$ matrix of regression coefficients, x_t is a $q \times 1$ vector of covariates, and Σ is the $p \times p$ covariance matrix. This model can be written as

$$y = (X \otimes I_p)\beta + e, \tag{6.2}$$

where $X = (x_1, \ldots, x_n)'$, $Y = (y_1, \ldots, y_n)$, $y = \text{vec}(Y)$, $e = \text{vec}(e_1, \ldots, e_n)$, $\beta = \text{vec}(B)$, and I_k is the $k \times k$ identity matrix. The variance covariance matrix of e is $I_n \otimes \Sigma$.

Suppose $\Sigma \sim HIW(g, \delta, \Phi)$ and let $M^+(g)$ be defined as in Section 2.5. Then $p(y|\Sigma, g) = p(y|\Sigma)I(\Omega \in M^+(g))$, where $I(\Omega \in M^+(g)) = 1$ if $\Omega \in M^+(g))$ and zero otherwise. The likelihood for this model is

$$\begin{aligned} p(y|\beta, \Sigma, g) &= |2\pi(I_n \otimes \Sigma)|^{-1/2} \exp(-1/2(y - (X \otimes I_p)\beta)' \\ &\qquad \times (I_n \otimes \Sigma^{-1})(y - (X \otimes I_p)\beta))I(\Omega \in M^+(g)) \qquad (6.3) \\ &= (2\pi)^{-np/2}|\Sigma|^{-n/2} \exp(-1/2E)I(\Omega \in M^+(g)), \qquad (6.4) \end{aligned}$$

where

$$\begin{aligned} E &= (y - (X \otimes I_p)\beta)'(I_n \otimes \Sigma^{-1})(y - (X \otimes I_p)\beta) \\ &= y'(I_n \otimes \Sigma^{-1})y - 2\beta'(X' \otimes \Sigma^{-1})y + \beta'(X'X \otimes \Sigma^{-1})\beta. \end{aligned} \tag{6.5}$$

6.2.2 Prior for β and Ω

We take the prior for β given Σ as $\beta|\Sigma \sim N(0, c((X'X)^{-1} \otimes \Sigma))$. This is a scaled version of the likelihood as a function of β, but is centred at zero.

Let $\gamma = (\gamma_1, \ldots, \gamma_q)$ be a vector of binary variables such that the ith variable of x_t is included in the regression if $\gamma_i = 1$ and excluded if $\gamma_i = 0$. Let $q_\gamma = \sum_{i=1}^q \gamma_i$. Given γ, $y = (X_\gamma \otimes I_p)\beta_\gamma + e$ and $\beta_\gamma \sim N(0, c(X_\gamma' X_\gamma)^{-1} \otimes \Sigma)$, and hence

$$\begin{aligned} p(\beta_\gamma|\Sigma, g, c) &= |2\pi c((X_\gamma' X_\gamma)^{-1} \otimes \Sigma)|^{-1/2} \exp(-(2c)^{-1}\beta_\gamma'(X_\gamma' X_\gamma \otimes \Sigma^{-1})\beta_\gamma)I(\Omega \in M^+(g)) \\ &= (2\pi)^{-pq_\gamma/2} c^{-pq_\gamma/2} |X_\gamma' X_\gamma|^{p/2} |\Sigma|^{-q_\gamma/2} \\ &\qquad \times \exp(-(2c)^{-1}\beta_\gamma'(X_\gamma' X_\gamma \otimes \Sigma^{-1})\beta_\gamma)I(\Omega \in M^+(g)). \end{aligned} \tag{6.6}$$

Conditional on (γ, Σ, g) the joint distribution $p(y, \beta_\gamma|\gamma, \Sigma, g)$ is

$$p(y|\beta_\gamma, \gamma, \Sigma, g)p(\beta_\gamma|\gamma, \Sigma, g) = (2\pi)^{-(n+q_\gamma)p/2}|\Sigma|^{-(n+q_\gamma)/2}|X'_\gamma X_\gamma|^{p/2} \times c^{-pq_\gamma/2}\exp(-E/2)I(\Omega \in M^+(g)), \tag{6.7}$$

where

$$\begin{aligned} E &= \beta'_\gamma(X'_\gamma \otimes \Sigma^{-1})\beta_\gamma(1+1/c) - 2\beta'_\gamma(X'_\gamma X_\gamma \otimes \Sigma^{-1})y + y'(I_p \otimes \Sigma^{-1})y \\ &= (\beta_\gamma - \widehat{\beta}_\gamma)'(X'_\gamma X_\gamma \otimes \Sigma^{-1})(\beta_\gamma - \widehat{\beta}_\gamma)(1+1/c) \qquad (6.8) \\ &\quad + y'(I_n \otimes \Sigma^{-1})y - \widehat{\beta}'_\gamma(X'_\gamma X_\gamma \otimes \Sigma^{-1})\widehat{\beta}_\gamma(1+1/c) \qquad (6.9) \end{aligned}$$

and

$$\widehat{\beta}_\gamma = c/(1+c)((X'_\gamma X_\gamma)^{-1}X'_\gamma) \otimes I_n)y. \tag{6.10}$$

Let $H_\gamma = X_\gamma(X'_\gamma X_\gamma)^{-1}X'_\gamma$. Then

$$\widehat{\beta}'_\gamma(X'_\gamma X_\gamma \otimes \Sigma^{-1})\widehat{\beta}_\gamma(1+1/c) = (c/(1+c))y'(H_\gamma \otimes \Sigma^{-1})y, \tag{6.11}$$

and so

$$E = (\beta_\gamma - \widehat{\beta}_\gamma)'(X'_\gamma X_\gamma \otimes \Sigma^{-1})(\beta_\gamma - \widehat{\beta}_\gamma)(1+1/c) + y'((I_n - c/(1+c)H_\gamma) \otimes \Sigma^{-1})y. \tag{6.12}$$

Using these expressions, the marginal likelihood of y given Σ, g and γ can be expressed as

$$\begin{aligned} p(y|\gamma, \Sigma, g) &= \int p(y|\beta_\gamma, \Sigma, g)p(\beta_\gamma|\Sigma, g)d\beta_\gamma \\ &= (2\pi)^{-(n+q_\gamma)p/2}|\Sigma|^{-(n+q_\gamma)/2}|X'_\gamma X_\gamma|^{p/2}c^{-pq_\gamma/2}\exp(-1/2y'(M_\gamma \otimes \Omega)y) \\ &\quad \times \int \exp\left(-1/2\left(\frac{1+c}{c}\right)(\beta_\gamma - \widehat{\beta}_\gamma)'(X'_\gamma X_\gamma \otimes \Sigma^{-1})(\beta_\gamma - \widehat{\beta}_\gamma)\right)d\beta_\gamma \\ &\quad \times \left|(2\pi)^{-1}\left(\frac{1+c}{c}\right)(X'_\gamma X_\gamma \otimes \Sigma^{-1})\right|^{1/2}\left|(2\pi)^{-1}\left(\frac{1+c}{c}\right)(X'_\gamma X_\gamma \otimes \Sigma^{-1})\right|^{-1/2} \\ &\quad \times I(\Omega \in M^+(g)) \\ &= (2\pi)^{-(n+q_\gamma)p/2}|\Sigma|^{-(n+q_\gamma)/2}|X'_\gamma X_\gamma|^{p/2}c^{-pq_\gamma/2}\exp(-1/2y'(M_\gamma \otimes \Omega)y) \\ &\quad \times (2\pi)^{pq_\gamma/2}\left(\frac{1+c}{c}\right)^{-pq_\gamma/2}|X'_\gamma X_\gamma|^{-p/2}|\Sigma|^{q_\gamma/2}I(\Omega \in M^+(g)). \end{aligned} \tag{6.13}$$

where $M_\gamma = I_n - c/(1+c)H_\gamma$. Substituting Ω for Σ^{-1} in (6.13) gives

$$p(y|\gamma, \Omega, g) = (2\pi)^{-np/2}|\Omega|^{n/2}(1+c)^{-pq_\gamma/2}\exp(-1/2y'(M_\gamma \otimes \Omega)y)I(\Omega \in M^+(g)). \tag{6.14}$$

We need the following lemma, whose proof is straightforward.

Lemma 6.2.1 *Suppose that A is an $r \times s$ matrix and B is $s \times r$. Then*

$$traceAB = (vecA)'vec(B') \tag{6.15}$$
$$= (vec(A'))'vec(B). \tag{6.16}$$

Using Lemma 6.2.1,

$$\begin{aligned} y'(M_\gamma \otimes \Omega)y &= \text{vec}(Y)'\text{vec}(\Omega Y M_\gamma) \\ &= \text{vec}(Y')'\text{vec}(\Omega Y M_\gamma \\ &= \text{trace}(Y M_\gamma Y'\Omega) \qquad (6.17) \\ &= \text{trace}(S(\gamma)\Omega), \qquad (6.18) \end{aligned}$$

where $S(\gamma) = Y(I - (c/1+c)H_\gamma)Y'$.

Hence

$$p(y|\gamma, \Omega, g) \propto |\Omega|^{n/2}(1+c)^{-pq_\gamma/2}\text{etr}(-1/2S(\gamma)\Omega). \tag{6.19}$$

Because $\Omega \sim HW(g, \delta, \Phi)$, or, equivalently, $\Sigma \sim HIW(g, \delta, \Phi)$,

$$p(\Omega|g, \delta, \Phi) = h(g, \delta, \Phi)|\Omega|^{(\delta-2)/2}\text{etr}(-1/2\Phi\Omega),$$

where $h(g, \delta, \Phi)$ is the normalising constant in (4.5) for the HIW distribution given in Section 4.2. Therefore, we can derive an expression for $p(y|\gamma, g, \delta, \Phi)$ on which the reduced conditional sampler is based. In the following derivation we use that γ is a priori independent of $\{\Omega, g\}$. In particular, $p(\gamma|g, \Omega) = p(\gamma)$, $p(\Omega|g, \gamma) = p(\Omega|g)$ and $p(g|\gamma) = p(g)$.

$$\begin{aligned} p(y|\gamma, g, \delta, \Phi) &= p(y|\gamma, \Omega, g, \delta, \Phi)p(\Omega|\gamma, g, \delta, \Phi)/p(\Omega|y, \gamma, g, \delta, \Phi) \\ &= \left[2\pi^{-np/2}(1+c)^{-pq_\gamma/2}|\Omega|^{n/2}\text{etr}(-1/2S(\gamma)\Omega)h(g, \delta, \Phi)|\Omega|^{(\delta-2)/2}\text{etr}(-1/2\Phi\Omega)\right] \qquad (6.20) \\ &\quad / \left[h(g, \delta+n, \Phi+S(\gamma))|\Omega|^{(n+\delta-2)/2}\text{etr}(-1/2(\Phi+S(\gamma))\Omega)\right] \\ &= 2\pi^{-np/2}(1+c)^{-pq_\gamma/2}h(g, \delta, \Phi)/h(g, \delta+n, \Phi+S(\gamma)). \qquad (6.21) \end{aligned}$$

6.2.3 Prior for the vector of binary indicator variables

The prior is described in Section 5.2.3.

6.2.4 Permanently selected variables

The model for permanently selected variables is described in Section 5.2.4.

6.2.5 Priors for Σ, Φ, g

The prior is described in Section 5.2.5.

6.3 Sampling scheme

This section describes the decomposable sampling scheme for the regression model with γ, g and the parameters in Φ generated using the following Markov chain Monte Carlo sampling scheme.

1. $\gamma|y, x, \gamma_{-i}, g, \delta, \Phi$;
2. $\rho, \tau|y, x, \gamma, g, \delta, \Phi$;
3. $g|y, x, \gamma, \delta, \Phi$.

We now give details of steps 1 and 3. In step 1, similarly to Section 4.9, let $\gamma_{-i} = \{\gamma_k, k \neq i\}$ and write $\gamma = (\gamma_i, \gamma_{-i})$. Since $p(\gamma|y, \Omega, g, \delta, \Phi) \propto p(y|\gamma, g, \delta, \Phi)p(\gamma)$, γ can be sampled from its posterior distribution using Markov chain Monte Carlo and (6.21) by generating the γ_i one at a time, conditional on g, δ, Φ and γ_{-i}, using the following MH sampling scheme. For $i = 1, \ldots, p$, let $\gamma = (\gamma_i, \gamma_{-i})$ and let $\gamma^c = (\gamma_1^c, \ldots \gamma_q^c)'$ be the current value of γ, given the generated values $\gamma_1, \ldots, \gamma_{i-1}$ generated thus far. Define the proposal γ^p (conditional on γ^c) as $\gamma^p = (\gamma_i^p, \gamma_{-i}^c)$, where $\gamma_i^p = 1 - \gamma_i^c$. This means that the proposal density for γ_i is $q_\gamma(a|b, \gamma_{-i}^c)$ where a and b are each either 0 or 1, and $q_\gamma(a = 1 - b|b, \gamma_{-ij}^c) = 1$. The MH acceptance probabilty for the proposal is therefore

$$\min\left\{1, \frac{p(y|\gamma^p, g, \delta, \Phi)}{p(y|\gamma^c, g, \delta, \Phi)} \frac{p(\gamma^p)}{p(\gamma^c)}\right\} \tag{6.22}$$

because $q_\gamma(\gamma_i^c|\gamma_i^p, \gamma_{-i}^c)/q_\gamma(\gamma_i^p|\gamma_i^c, \gamma_{-i}^c) = 1$. The ratio $p(\gamma^p)/p(\gamma^c)$ is known and the ratio of likelihoods $p(y|\gamma^p, g, \delta, \Phi)/p(y|\gamma^c, g, \delta, \Phi)$ is given by (6.21).

Similarly, $p(g|y, \gamma, \delta, \Phi) \propto p(y|\gamma, g, \delta, \Phi)p(g)$. Hence in step 3, we can sample $g = (e_{ij}, e_{-ij})$ from its posterior conditional on $\{\gamma, \delta, \Phi\}$ via the edge indicators using the same Metropolis Hastings Markov chain Monte Carlo sampling scheme described in Section 4.9, except that the likelihood of that section is replaced by (6.21). Note that we generate the graph g conditional on γ and hence the term $(1 + c)^{-pq_\gamma}$ in the likelihood (as a function of g) is a constant and drops out of the Metropolis Hastings ratio.

Both β and Ω are generated in order to compare performance with the estimates in Cripps et al. (2005) and in Chapter 5. Ω is sampled from $p(\Omega|y, g, \gamma)$ as described in

Sections 4.3 and 4.11, but with S_y replaced by $S(\gamma)$. The coefficient vector β_γ is sampled from $p(\beta_\gamma|y, g, \gamma, \Omega)$ using (5.6) of Section 5.2.2. However, generating β and Ω does not affect the convergence and mixing of the sampling scheme introduced in this chapter.

The sampling scheme described in this chapter is different to the scheme described in Section 5.3. Both Cripps et al. (2005) and the sampling scheme in Chapter 5 generate the elements of γ one at a time by calculating

$$p(\gamma_i = 1|y, \Omega, \gamma_{-i})$$

which depends on Ω. On the other hand, in the reduced conditional scheme proposed in this chapter, γ and g are sampled from their posteriors without conditioning on Ω by using Markov chain Monte Carlo and (6.21).

6.4 Results

Let $DCPCSVSNB$ denote the decomposable model described in this chapter with columns of the B matrix either dropped or retained and with selection on the graph $g(\Omega)$, where $g(\Omega)$ is restricted to be decomposable. It is convenient for $DCPCSVSNB$ to stand also for the sampling scheme used to estimate the model. The inverse covariance matrix Ω and the matrix of coefficients B are also generated, although not as part of the sampling scheme. In this chapter we use the cow data and the physical measurement data sets to compare $DCPCSVSNB$ to the $NDPCSVS$ and $DCPCSVS$ models. In order to make the comparison consistent, columns of the B matrix are also either dropped or retained in the $NDPCSVS$ and $DCPCSVS$ models used in this chapter.

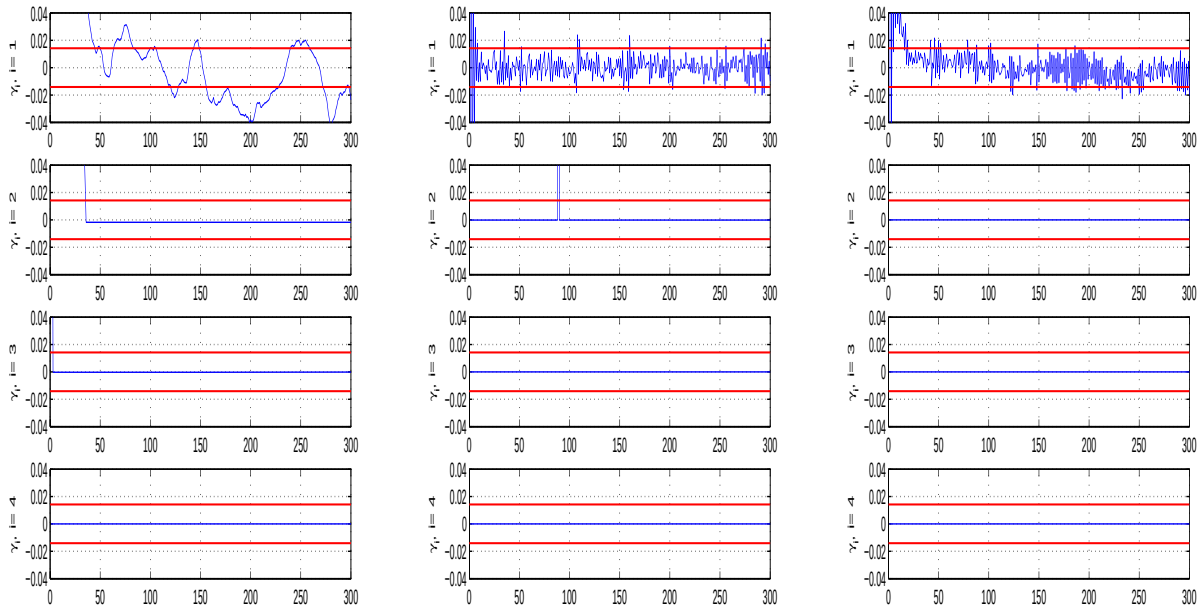
Let τS_y and $\tau S(\gamma)$ denote the scaled sum of squares form of Φ for models $DCPSVS$ and $DCPCSVSNB$, respectively. Let C be the partial correlations matrix. For $j = 1, \ldots, p$ and $i < j$, we define the binary variable $J_{ij} = 0$ if C_{ij} is identically zero and $J_{ij} = 1$ otherwise. Let $J = \{J_{ij}, i < j, j = 1, \ldots, p\}$. These binary variables are analogous to the γ_i binary variables that we use for variable selection, and in the decomposable case, J_{ij} are the edge indicators e_{ij}.

For the variable selection, we report the posterior means and standard errors of the regression coefficients, and the posterior probabilities of including a predictor variable in the regression. For the covariance selection, we report image plots of the estimates of C, J and Ω which are computed as the average of the iterates and which we we refer to as $\widehat{C}, \widehat{J}$ and $\widehat{\Omega}$, respectively. The image plots are lighter where the matrix is sparser.

6.4.1 Cow diet data

This section compares the $DCPCSVSNB$ model to the $NDPCSVS$ and $DCPCSVS$ models on the same cow diet dataset described in Section 5.4.2.

Figures 6.1 and 6.2 compare the autocorrelations of the iterates. Figure 6.1 shows the autocorrelations in the γ_i for each model. Panel (a) is for model $NDPCSVS$, and Panels (b) and (c) are for models $DCPCSVS$ and $DCPCSVSNB$, respectively, using the equicorrelated form of Φ. The autocorrelations of the iterates decay most rapidly to zero for the two decomposable models, suggesting that the greatest gains in efficiency are from reduced conditional covariance selection. Comparing the $NDPCSVS$ plots with the two decomposable model plots suggest that there is minimal additional efficiency gained from the reduced conditional variable selection sampler that is not already gained in the reduced conditional covariance selection sampler. The autocorrelations for the decomposable samplers using the τS_y form of Φ were similar to those for the equicorrelated form of Φ. The plots are omitted for brevity.



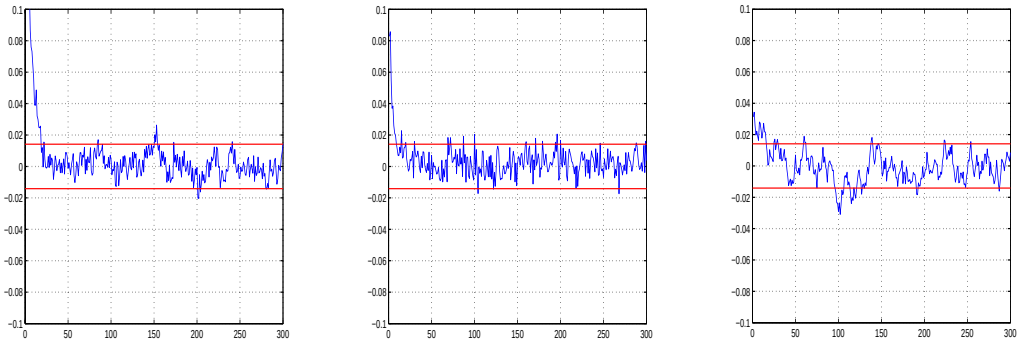
(a) (b) (c)

Figure 6.1: Autocorrelations of the iterates of the γ_i for the cow diet dataset. Panel (a) plots the autocorrelations for $NDPCSVS$. Panel (b) is the same for $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is the same for $DCPCSVSNB$ using the equicorrelated form of Φ.

Figure 6.2 compares the autocorrelations in the iterates of the log of the likelihood $p(y|g,\gamma,\Omega)$ for models $NDPCSVS$ and $DCPSCVS$, and in the iterates of the log of the reduced conditional likelihood $p(y|\gamma,g)$ for the $DCPCSVSNB$ model. Each of the decomposable models are for the equicorrelated form of Φ. The autocorrelations decay most rapidly to zero for the $DCPCSVSNB$ model, followed by the $DCPCSVS$ model. The autocorrelations in the log likelihood for $NDPCSVS$ are much better than for the γ_i, though not so good as the decomposable models. The autocorrelations for the decomposable models using the τS_y form of Φ are similar to the equicorrelated models, and the plots are omitted for brevity.

It is unclear why the autocorrelations for $DCPCSVSNB$ are not more significantly better than for $DCPCSVS$. We surmise that it is because both these use the same reduced conditional covariance selection sampler, and that the greatest gains in efficiency are from covariance selection rather than variable selection. That is, most of the autocorrelation is introduced in sampling Ω.

We conclude that the difference between $NDPCSVS$ and $DCPCSVS$ in dependence of the iterates is due to the greater efficiency of the $DCPCSVS$ sampling scheme which integrates out the covariance Σ; and secondly, that the difference between these two and the $DCPSCVSNB$ model in dependence of the iterates is due to the greater efficiency of the sampling scheme which integrates out both the covariance and regression coefficient parameters.



(a) (b) (c)

Figure 6.2: Autocorrelations of the iterates of the log likelihood. Panel (a) is the autocorrelations for $NDPCSVS$ of $\log(p(y|g,\gamma,\Omega)$. Panel (b) is the same for $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is the autocorrelations for for $DCPCSVSNB$ of $\log(p(y|g,\gamma)$ using the equicorrelated form of Φ.

The different model estimates of γ are compared in Table 6.1 which suggests that all model estimates are similar. The conclusion from the values of the γ_i is that *Initial weight* followed by *Level of diet additive* are significantly better predictors of the response variables than the remaining covariates. A full analysis of the complete interrelationships between variables could be done similarly to the analysis presented in Chapter 5.4 by considering the graphical pictures corresponding to the various estimates of $g(\Omega)$, but for brevity this analysis is omitted.

The different model estimates of the regression coefficients β are compared in Table 6.2 which suggests that the model estimates of $NDPCSVS$, $DCPCSVS$ and $DCPCSVSNB$ are similar.

	γ_0	γ_1	γ_2	γ_3	γ_4
$NDPCSVS$	NA	0.4934	0.0018	0.0003	1.0000
equi. $DCPCSVS$	NA	0.4572	0.0006	0	1.0000
τS_y $DCPCSVS$	NA	0.4572	0.0006	0	1.0000
eq. $DCPCSVSNB$	NA	0.4394	0	0	1.0000
$\tau S(\gamma)$ $DCPCSVSNB$	NA	0.4572	0.0006	0	1.0000

Table 6.1: Posterior mean estimates of $\gamma_1, \ldots, \gamma_4$ for the grouped cow diet dataset. See text for details. γ_0 is identically 1 as the coefficient is always included. Integer values are given whenever these are exact.

Figures 6.3 and 6.4 compare $\widehat{C}, \widehat{J}$ and $\widehat{\Omega}$ for models $NDPCSVS$, $DCPCSVS$ and $DCPCSVSNB$. The image plots of $\widehat{\Omega}$ of all models appear almost identical. The image plots of $\widehat{C}$ and $\widehat{J}$ for $DCPSCVS$ using the τS_y form of Φ are more similar to $NDPCSVS$ than the remaining decomposable models. However the image plots of $\widehat{J}$ and $\widehat{C}$ are almost identical in the last row, suggesting all models agree on the partial correlations between $(y_5, y_6)' = (Final\ weight, Milk\ protein(\%))$ and the remaining variables. There are other regions of distinct similarity, suggesting that overall the model estimates are very similar.

Posterior mean

	$\beta_{1,0}$	$\beta_{1,1}$	$\beta_{1,2}$	$\beta_{1,3}$	$\beta_{1,4}$	$\beta_{2,0}$	$\beta_{2,1}$	$\beta_{2,2}$	$\beta_{2,3}$	$\beta_{2,4}$
$NDPCSVS$	8.1667	0.5353	0.0001	-0.0000	0.0062	24.3243	-0.2539	0.0001	-0.0000	0.0271
τS_y $DCPCSVS$	8.1690	0.4196	0	0	0.0063	24.2531	-0.0866	0	0	0.0271
$equi$ $DCPCSVS$	8.1628	0.5150	0	-0.0000	0.0063	24.3140	-0.1221	0	0.0000	0.0271
τS_y $DCPCSVSNB$	8.1242	0.4944	0.0000	0	0.0063	24.2279	-0.1565	-0.0003	0	0.0272
$equi$ $DCPCSVSNB$	8.1652	0.4965	0	0	0.0063	24.2415	-0.1120	0	0	0.0272
	$\beta_{3,0}$	$\beta_{3,1}$	$\beta_{3,2}$	$\beta_{3,3}$	$\beta_{3,4}$	$\beta_{4,0}$	$\beta_{4,1}$	$\beta_{4,2}$	$\beta_{4,3}$	$\beta_{4,4}$
$NDPCSVS$	2.6396	0.9696	0.0001	0.0000	0.0006	8.5521	-0.1007	-0.0001	-0.0000	-0.0000
τS_y $DCPCSVS$	2.6661	0.7284	0	0	0.0006	8.5469	-0.0744	0	0	-0.0000
$equi$ $DCPCSVS$	2.6334	0.9485	0	-0.0000	0.0006	8.5538	-0.0995	0	-0.0000	-0.0000
τS_y $DCPCSVSNB$	2.6444	0.8927	0.0000	0	0.0006	8.5470	-0.0990	-0.0001	0	-0.0000
$equi$ $DCPCSVSNB$	2.6486	0.8671	0	0	0.0006	8.5544	-0.0850	0	0	-0.0000
	$\beta_{5,0}$	$\beta_{5,1}$	$\beta_{5,2}$	$\beta_{5,2}$	$\beta_{5,4}$	$\beta_{6,0}$	$\beta_{6,1}$	$\beta_{6,2}$	$\beta_{6,3}$	$\beta_{6,4}$
$NDPCSVS$	220.4607	-110.7841	-0.0013	-0.0001	0.8077	3.3412	-0.0828	0.0000	0.0000	-0.0001
τS_y $DCPCSVS$	216.8705	-83.4706	0	0	0.8073	3.3388	-0.0643	0	0	-0.0001
$equi$ $DCPCSVS$	219.4685	-108.9125	0	-0.0001	0.8082	3.3417	-0.0827	0	-0.0000	-0.0001
τS_y $DCPCSVSNB$	216.8705	-83.4706	0	0	0.8073	3.3388	-0.0643	0	0	-0.0001
$equi$ $DCPCSVSNB$	217.6527	-98.4224	0	0	0.8087	3.3424	-0.0720	0	0	-0.0001

Table 6.2: Comparison of posterior means of the regression coefficients for the grouped cow diet data. Models $NDPCSVS$ and $DCPCSVS$ for the τS_y and the equicorrelated forms of Φ, respectively, are compared to models $DCPCSVSNB$ for the τS_y and the equicorrelated forms of Φ, respectively. Integer values are given when these are exact.

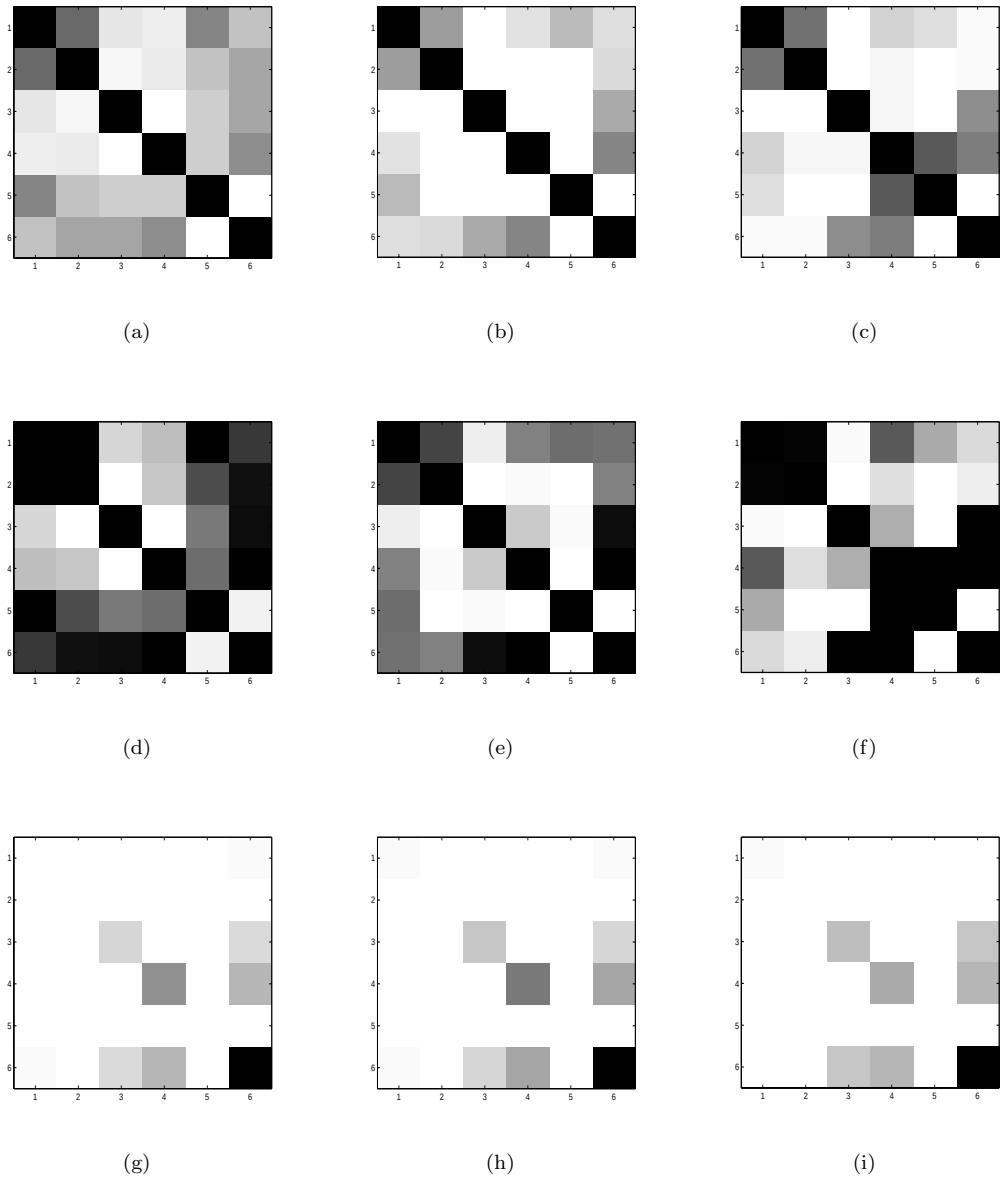
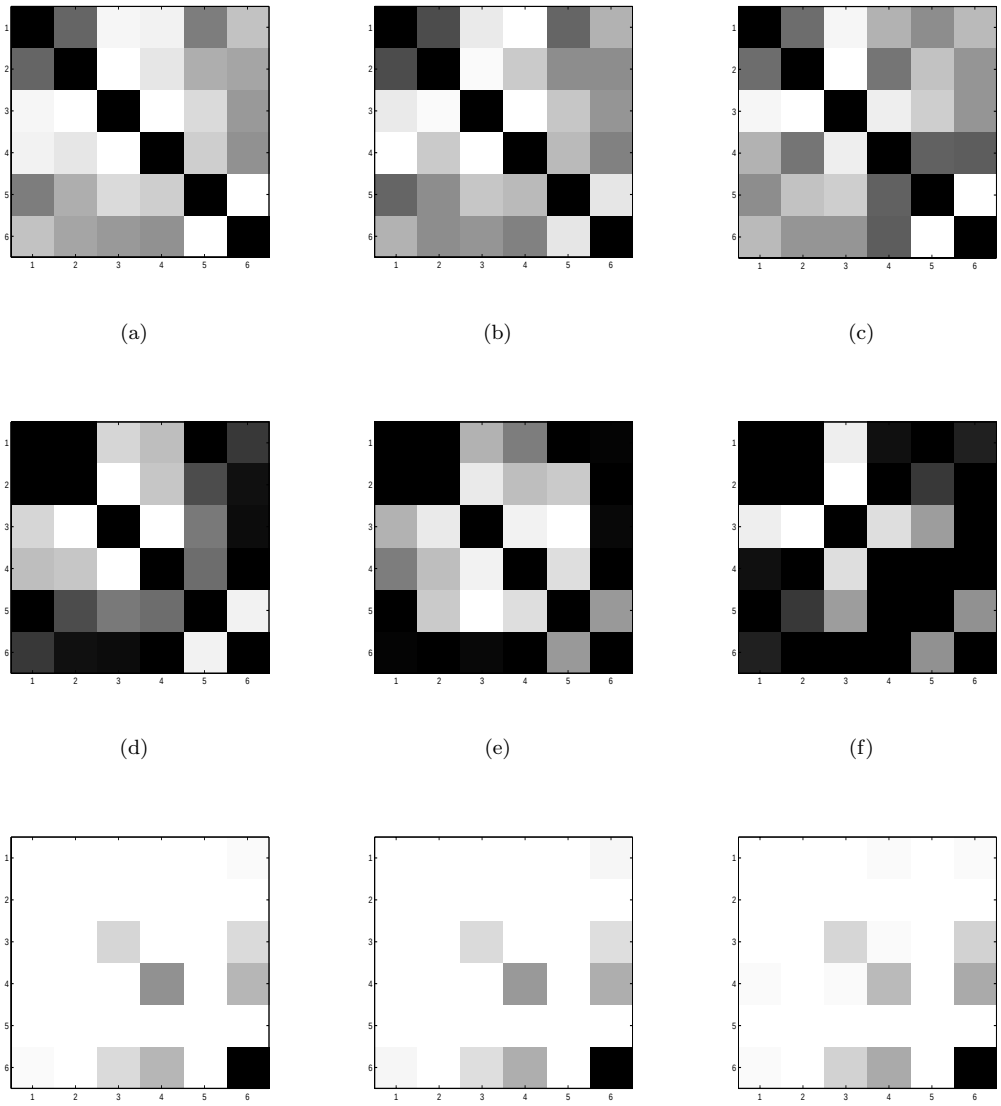


Figure 6.3: Image plots of $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$ for the cow diet dataset. Panels (a), (d) and (g) are $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$, respectively, for model $NDPCSVS$. Panels (b), (e) and (h) are the same for model $DCPCSVS$ using the equicorrelated form of Φ. Panels (c), (f) and (i) are the same for model $DCPCSVSNB$ using the equicorrelated form of Φ.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 6.4: Image plots of $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$ for the cow diet dataset. Panels (a), (d) and (g) are $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$, respectively, for model $NDPCSVS$. Panels (b), (e) and (h) are the same for model $DCPCSVS$ using the τS_y form of Φ. Panels (c), (f) and (i) are the same for model $DCPCSVSNB$ using the $\tau S_(\gamma)$ form of Φ.

6.4.2 Physical measurements data: model 2

This section compares the $DCPCSVSNB$ model to the $NDPCSVS$ and $DCPCSVS$ models on the same physical measurements dataset described in Section 5.4.3. The covariate predictor variables are *Mass* and *Height*. The remaining $p = 9$ variables are used as the response vector y. The $p = 9$ response variables are indexed in the following order:

r1 Fore: maximum circumference of forearm,

r2 Bicep: maximum circumference of bicep,

r3 Chest: distance around chest directly under the armpits,

r4 Neck: distance around neck, approximately halfway up,

r5 Shoulders: distance around shoulders, measured around the peak of the shoulder blades

r6 Waist: distance around waist, approximately trouser line,

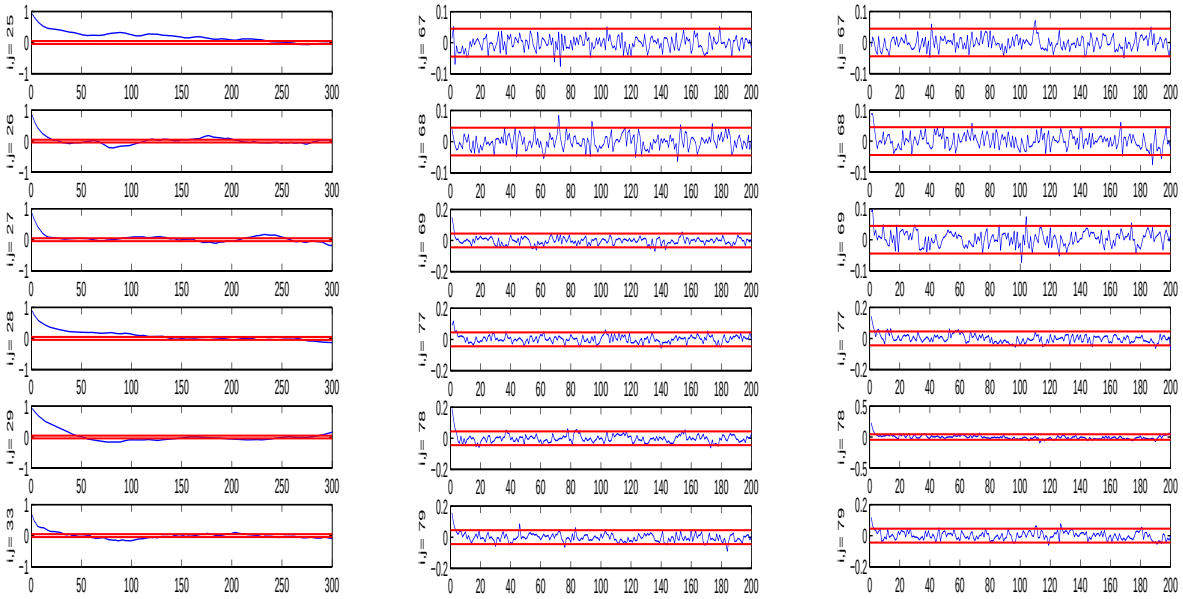
r7 Calf: maximum circumference of calf,

r8 Thigh: circumference of thigh, measured halfway between the knee and the top of the leg,

r9 Head: maximum circumference of head.

Figures 6.5 and 6.6 compare the autocorrelations in the iterates of a representative sample of the Ω_{ij}. Figure 6.5 compares $NDPCSVS$ with the decomposable samplers using the equicorrelated form of Φ. Figure 6.6 compares $NDPCSVS$ with the decomposable samplers using the τS_y form of Φ. The autocorrelations show the same trends between models as those reported in Section 6.4.1 for the cow diet data. The autocorrelations of the iterates decay least rapidly to zero for the $NDPCSVS$ model, but there appears to be no difference in the autocorrelations of the two decomposable samplers. The empirical evidence for the physical measurements data provides further support for the conclusions on the relative efficiency of the models given in Section 6.4.1. For brevity these conclusions are not repeated here.

The sampling average values of γ are compared in Table 6.3 which suggests that all model estimates are very similar. The value for γ_1 is identically one for every model, suggesting that the corresponding predictor variable *Mass* is a significantly better predictor of the response variables than *Height*. A full analysis of the complete interrelationships



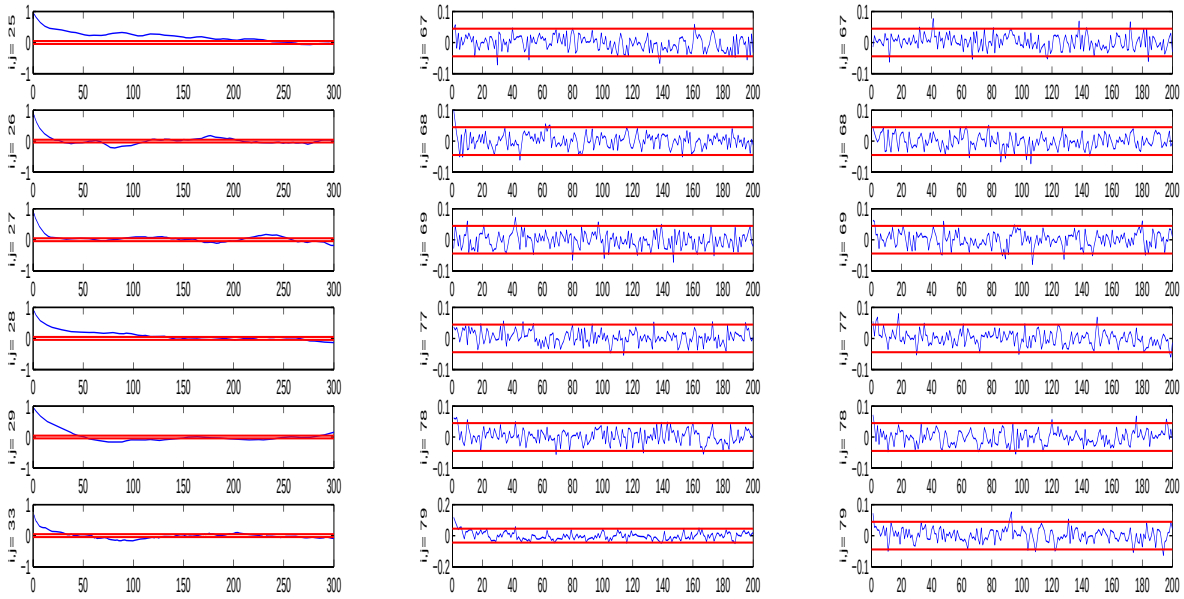
(a) (b) (c)

Figure 6.5: Autocorrelations of the iterates of the Ω_{ij} for a representative selection of Ω_{ij} and the physical measurements dataset. Panel (a) plots the autocorrelations for $NDPCSVS$. Panel (b) is the same for $DCPCSVS$ using the equicorrelated form of Φ. Panel (c) is the same for $DCPCSVSNB$ using the equicorrelated form of Φ.

between variables could be done similarly to the analysis presented in Chapter 5.4 by considering the graphical pictures corresponding to the various estimates of $g(\Omega)$, but for brevity this analysis is omitted.

	γ_0	γ_1	γ_2
$NDPCSVS$	NA	1.0000	0.0070
equicorrelated $DCPCSVS$	NA	1.0000	0.0027
τS_y $DCPCSVS$	NA	1.0000	0.0118
equicorrelated $DCPCSVSNB$	NA	1.0000	0.0035
$\tau S(\gamma)$ $DCPCSVSNB$	NA	1.0000	0.0026

Table 6.3: Posterior mean estimates of γ_1, γ_2 corresponding to *Mass* and *Height*, respectively, for the physical measurements dataset using Model 2. See text for details. γ_0 is identically 1 as the coefficient is always included. Integer values are given whenever these are exact.



(a) (b) (c)

Figure 6.6: Autocorrelations of the iterates of the Ω_{ij} for a representative selection of Ω_{ij} and the physical measurements dataset. Panel (a) plots the autocorrelations for $NDPCSVS$. Panel (b) is the same for $DCPCSVS$ using the τS_y form of Φ. Panel (c) is the same for $DCPCSVSNB$ using the $\tau S(\gamma)$ form of Φ.

The different model estimates of the regression coefficients β are compared in Table 6.4 which suggests that the model estimates of $NDPCSVS$, $DCPCSVS$ and $DCPCSVSNB$ are similar.

Figures 6.7 and 6.8 compare $\widehat{C}, \widehat{J}$ and $\widehat{\Omega}$ for models $NDPCSVS$, $DCPCSVS$ and $DCPCSVSNB$. Figure 6.7 is the decomposable estimates for the equicorrelated form of Φ, whilst Figure 6.8 is for the τS_y form of Φ. The image plots of $\widehat{C}$ and $\widehat{\Omega}$ for all models appear to be more similar than the image plots for $\widehat{J}$. However, in general, there are regions of distinct similarity in all the estimates which suggests that the efficiency gains of the reduced conditional decomposable samplers are not at the expense of the accuracy of their estimates.

Posterior mean

	$\beta_{1,0}$	$\beta_{1,1}$	$\beta_{1,2}$	$\beta_{2,0}$	$\beta_{2,1}$	$\beta_{2,2}$	$\beta_{3,0}$	$\beta_{3,1}$	$\beta_{3,2}$
$NDPCSVS$	15.4898	0.1504	-0.0002	18.3009	0.1850	-0.0012	61.8963	0.4561	-0.0017
τS_y $DCPCSVS$	15.5446	0.1504	-0.0006	18.6326	0.1826	-0.0021	62.4531	0.4525	-0.0034
$equi$ $DCPCSVS$	15.4444	0.1506	-0.0001	18.1987	0.1847	-0.0005	61.6513	0.4572	-0.0007
τS_y $DCPCSVSNB$	15.5048	0.1499	-0.0002	18.2608	0.1837	-0.0006	61.8561	0.4541	-0.0010
$equi$ $DCPCSVSNB$	15.4180	0.1512	-0.0001	18.2153	0.1849	-0.0007	61.6812	0.4574	-0.0010
	$\beta_{4,0}$	$\beta_{4,1}$	$\beta_{4,2}$	$\beta_{5,0}$	$\beta_{5,1}$	$\beta_{5,2}$	$\beta_{6,0}$	$\beta_{6,1}$	$\beta_{6,2}$
$NDPCSVS$	26.3929	0.1293	0.0000	71.5318	0.4776	0.0007	35.9504	0.6233	-0.0011
τS_y $DCPCSVS$	26.4010	0.1298	-0.0003	71.5503	0.4788	0.0001	36.0952	0.6247	-0.0025
$equi$ $DCPCSVS$	26.3587	0.1298	0.0000	71.5433	0.4782	0.0003	35.8419	0.6234	-0.0005
τS_y $DCPCSVSNB$	26.4407	0.1288	-0.0002	71.8285	0.4746	-0.0002	36.1118	0.6200	-0.0008
$equi$ $DCPCSVSNB$	26.3447	0.1303	-0.0001	71.5694	0.4786	0.0001	35.8697	0.6236	-0.0006
	$\beta_{7,0}$	$\beta_{7,1}$	$\beta_{7,2}$	$\beta_{8,0}$	$\beta_{8,1}$	$\beta_{8,2}$	$\beta_{9,0}$	$\beta_{9,1}$	$\beta_{9,2}$
$NDPCSVS$	22.7393	0.1755	0.0000	28.1495	0.2642	-0.0007	53.5506	0.0269	0.0000
τS_y $DCPCSVS$	22.8003	0.1755	-0.0003	28.4224	0.2625	-0.0015	53.5503	0.0281	-0.0005
$equi$ $DCPCSVS$	22.7346	0.1756	-0.0000	28.0614	0.2643	-0.0002	53.4823	0.0277	0.0001
τS_y $DCPCSVSNB$	22.8110	0.1748	-0.0002	28.1579	0.2632	-0.0004	53.6278	0.0259	-0.0002
$equi$ $DCPCSVSNB$	22.6997	0.1763	-0.0001	28.0609	0.2648	-0.0004	53.4831	0.0281	-0.0000

Table 6.4: Comparison of posterior means of the regression coefficients for the grouped physical measurements data, model 2. Models $NDPCSVS$ and $DCPCSVS$ for the τS_y and the equicorrelated forms of Φ, respectively, are compared to models $DCPCSVSNB$ for the τS_y and the equicorrelated forms of Φ, respectively. Integer values are given when these are exact.

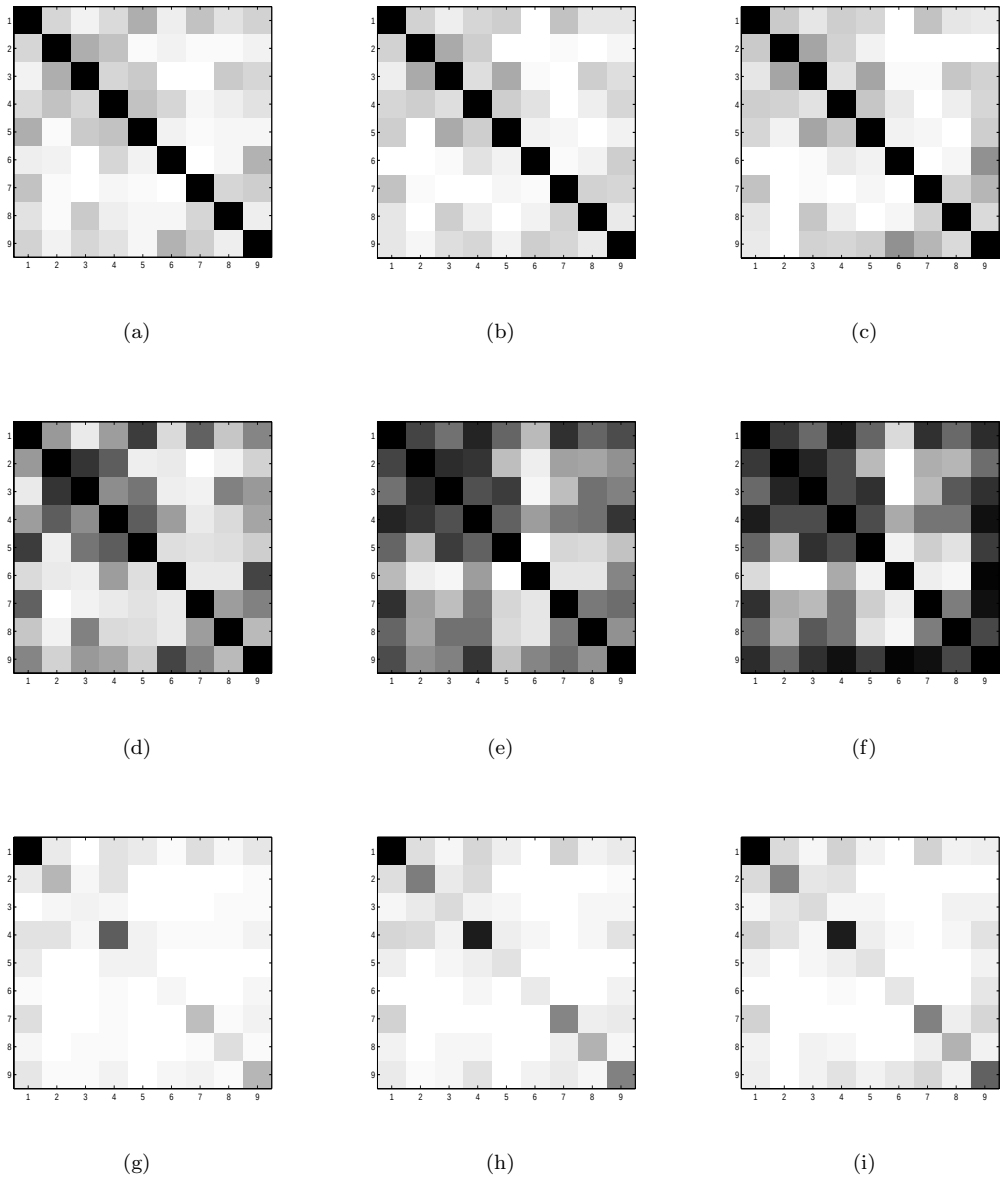


Figure 6.7: Image plots of $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$ for the physical measurements dataset. Panels (a), (d) and (g) are $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$, respectively, for model $NDPCSVS$. Panels (b), (e) and (h) are the same for model $DCPCSVS$ using the equicorrelated form of Φ. Panels (c), (f) and (i) are the same for model $DCPCSVSNB$ using the equicorrelated form of Φ.

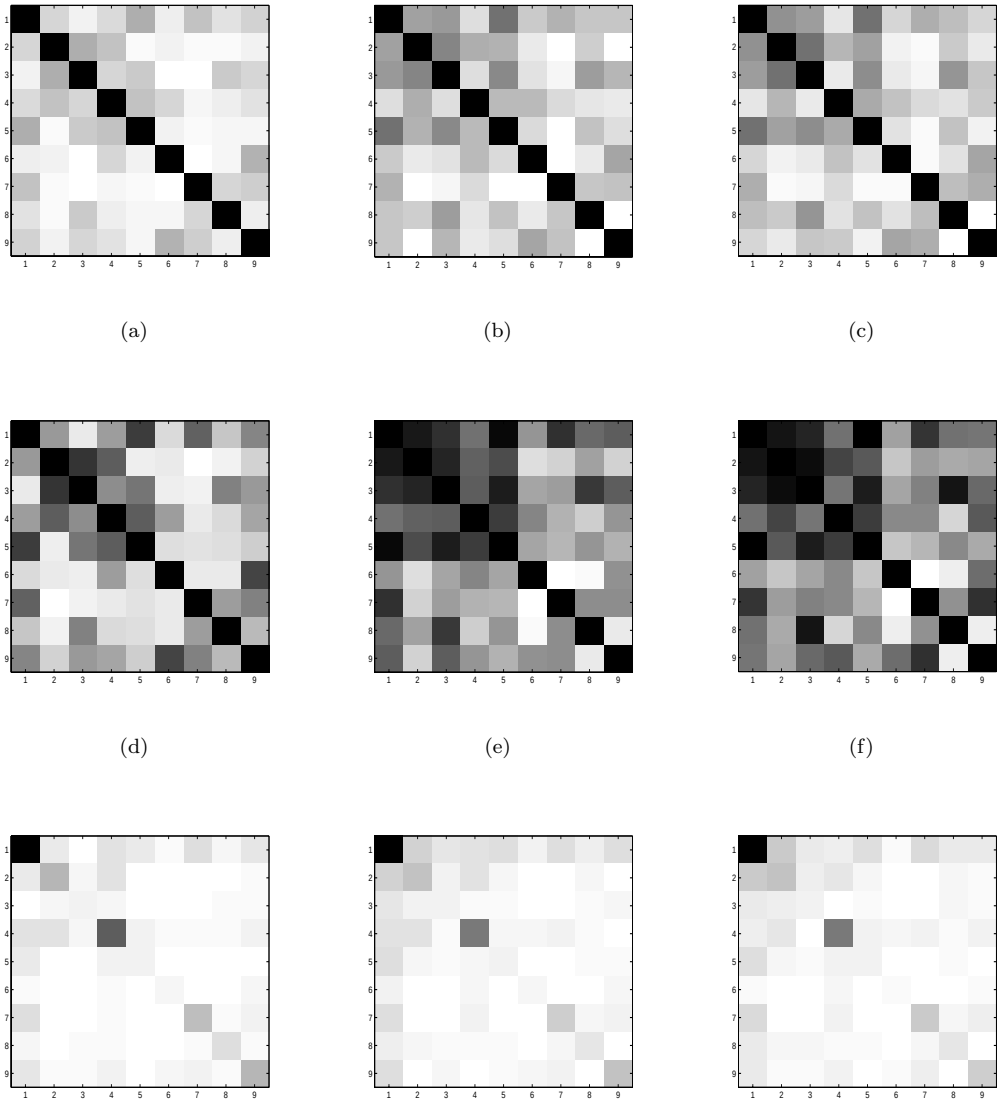


Figure 6.8: Image plots of $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$ for the physical measurements dataset. Panels (a), (d) and (g) are $\widehat{C}$, $\widehat{J}$ and $\widehat{\Omega}$, respectively, for model $NDPCSVS$. Panels (b), (e) and (h) are the same for model $DCPCSVS$ using the τS_y form of Φ. Panels (c), (f) and (i) are the same for model $DCPCSVSNB$ using the τS_y form of Φ.

6.4.3 Physical measurements data: model 3

In this section we compare the $DCPCSVSNB$ model to the $NDPCSVS$ and $DCPCSVS$ models on the same physical measurements dataset described in Section 5.4.3 but assuming a regression model in which the covariate predictor variables and the response variables of Section 6.4.2 are interchanged. That is, there are 9 predictor variables and $p = 2$ response variables, and these are indexed in the following order:

p1 Fore: maximum circumference of forearm,

p2 Bicep: maximum circumference of bicep,

p3 Chest: distance around chest directly under the armpits,

p4 Neck: distance around neck, approximately halfway up,

p5 Shoulders: distance around shoulders, measured around the peak of the shoulder blades,

p6 Waist: distance around waist, approximately trouser line,

p7 Calf: maximum circumference of calf,

p8 Thigh: circumference of thigh, measured halfway between the knee and the top of the leg,

p9 Head: maximum circumference of head,

r1 Mass: weight in kg, and

r2 Height: height in cm.

The trends in the autocorrelations of γ_i, log likelihood and Ω_{ij} iterates are the same as those reported in Subsections 6.4.1 and 6.4.2, providing more empirical evidence that the decomposable samplers are more efficient than the nondecomposable sampler, and that $DCPCSVSNB$ is the most efficient sampler. The plots of the autocorrelations are omitted for brevity.

The sampling average values of γ are compared in Table 6.5 which suggests that all model estimates are similar. The relative values of the γ_i, $, i = 1, \ldots, 9$ for all models suggests that *Fore* and *Waist* (corresponding to γ_1 and γ_6, respectively) are the two predictors with highest probability of selection in the model. γ_8, corresponding to *Thigh* is the predictor with next highest posterior mean of inclusion in the model. A full analysis

of the complete interrelationships between variables can be done similarly to the analysis presented in Chapter 5.4 by considering the graphical pictures corresponding to the various estimates of $g(\Omega)$, but for brevity this analysis is omitted.

	γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7	γ_8	γ_9
$NDPCSVS$	NA	0.8074	0.0911	0.0486	0.0380	0.0917	0.9827	0.1380	0.1527	0.0549
equi. $DCPCSVS$	NA	0.6162	0.0936	0.0633	0.0483	0.1030	0.8264	0.1040	0.1458	0.0302
τS_y $DCPCSVS$	NA	0.7218	0.1191	0.0639	0.0616	0.0793	0.8536	0.1178	0.1840	0.0556
eq. $DCPCSVSNB$	NA	0.6092	0.0624	0.0512	0.0425	0.0523	0.9302	0.1016	0.1323	0.0506
$\tau S(\gamma)$ $DCPCSVSNB$	NA	0.8581	0.0565	0.0473	0.0315	0.0377	0.9950	0.0790	0.1309	0.0519

Table 6.5: Posterior mean estimates of $\gamma_1, \ldots, \gamma_9$ for the physical measurements dataset using Model 3. See text for details. γ_0 is identically 1 as the coefficient is always included. Integer values are given whenever these are exact.

The different model estimates of the regression coefficients β are compared in Table 6.6 which suggests that the model estimates of $NDPCSVS$, $DCPCSVS$ and $DCPCSVSNB$ are similar.

In terms of which variables to select in the model, there are two points to note. First, in Table 6.5 the values of γ_1 and γ_6 are similar, and both significantly greater than the remaining γ_i. Second, the remaining γ_i are all very similar. These values suggest that the regression model should include the covariates *Waist* and *Fore*, and that the remaining covariates are not adding much to the model.

In the model of this section, there are only $p = 2$ response variables, so the matrices J and Ω are each 2×2. Figure 6.7 compares $\widehat{J}$ and $\widehat{\Omega}$. The three models $NDPCSVS$, $DCPCSVS$ and $DCPCSVSNB$ have almost identical estimates of J. In general, the entries in the table suggest that all models provide similar estimates.

Posterior mean

	$\beta_{1,0}$	$\beta_{1,1}$	$\beta_{1,2}$	$\beta_{1,3}$	$\beta_{1,4}$
$NDPCSVS$	-64.7888	2.0719	-0.0087	0.0055	0.0150
τS_y $DCPCSVS$	-83.9945	2.4310	-0.2015	0.0284	0.0772
$equi$ $DCPCSVS$	-63.0667	1.7753	0.0469	0.0133	0.0462
τS_y $DCPCSVSNB$	-64.1581	2.2350	0.0029	0.0051	0.0060
$equi$ $DCPCSVSNB$	-59.1494	1.6695	0.0279	0.0129	0.0270
	$\beta_{1,5}$	$\beta_{1,6}$	$\beta_{1,7}$	$\beta_{1,8}$	$\beta_{1,9}$
$NDPCSVS$	0.0355	0.7474	0.1628	0.0994	-0.0310
τS_y $DCPCSVS$	0.0284	0.5861	0.1465	0.5773	0.0206
$equi$ $DCPCSVS$	0.0727	0.6884	0.1948	0.1334	-0.0200
τS_y $DCPCSVSNB$	0.0102	0.7571	0.0967	0.0824	-0.0286
$equi$ $DCPCSVSNB$	0.0267	0.7938	0.1476	0.1144	-0.0309
	$\beta_{2,0}$	$\beta_{2,1}$	$\beta_{2,2}$	$\beta_{2,3}$	$\beta_{2,4}$
$NDPCSVS$	136.7393	0.5234	-0.1251	-0.0111	0.0444
τS_y $DCPCSVS$	116.4333	0.5145	-0.7215	0.0478	0.0862
$equi$ $DCPCSVS$	138.4147	0.4299	-0.0632	-0.0172	0.0169
τS_y $DCPCSVSNB$	137.5855	0.6275	-0.0759	-0.0120	0.0292
$equi$ $DCPCSVSNB$	139.8233	0.3725	-0.0480	-0.0071	0.0300
	$\beta_{2,5}$	$\beta_{2,6}$	$\beta_{2,7}$	$\beta_{2,8}$	$\beta_{2,9}$
$NDPCSVS$	0.0531	0.1456	0.0988	-0.0046	0.0202
τS_y $DCPCSVS$	0.0365	0.1389	0.1283	0.7946	-0.0697
$equi$ $DCPCSVS$	0.0628	0.1668	0.0709	0.0030	-0.0102
τS_y $DCPCSVSNB$	0.0162	0.1576	0.0765	-0.0197	0.0194
$equi$ $DCPCSVSNB$	0.0294	0.1727	0.0778	-0.0035	0.0177

Table 6.6: Comparison of posterior means of the regression coefficients for the grouped physical measurements data, model 3. Models $NDPCSVS$ and $DCPCSVS$ for the τS_y and the equicorrelated forms of Φ, respectively, are compared to models $DCPCSVSNB$ for the τS_y and the equicorrelated forms of Φ, respectively. Integer values are given when these are exact.

	$\widehat{J}_{1,2}$	$\widehat{\Omega}_{1,1}$	$\widehat{\Omega}_{1,2}$	$\widehat{\Omega}_{2,2}$
$NDPCSVS$	0.9807	0.1164	-0.0350	0.0235
equi. $DCPCSVS$	0.97782	0.1032	-0.0343	0.0248
$\tau S(\gamma)$ $DCPCSVS$	0.9932	0.1228	-0.0410	0.0275
equi. $DCPCSVSNB$	1	0.0939	-0.0364	0.0218
$\tau S(\gamma)$ $DCPCSVSNB$	1	0.1176	-0.0401	0.0212

Table 6.7: Estimates of J and Ω for all models. Both $\widehat{J}_{1,2}$ for models $DCPCSVS$ and $DCPCSVSNB$ are identically one.

6.5 Comparision to variable selection using leaps function

This section illustrates that graphical techniques provide more information for variable selection than standard linear regression techniques, by comparing the graphical analysis with the summary analysis from Larner, M. (1996) that is reproduced at the end of Subsection 5.4.3. The summary analysis is the output of the *leap* function method of variable selection which ranks the models in terms of the Mallow's Cp criterion, where the model with the lowest Cp score is judged to fit the data best. The standard regression analysis is based on successively choosing different combinations of predictor variables and calculating the Cp value of the corresponding models. The output reproduced at the end of Subsection 5.4.3 gives the lowest Cp score to the model which regresses *Mass* on the 4 variables *Fore, Waist, Height* and *Thigh*. In the graphical analysis in Subsection 6.4.3, *Height* is included as a response variable. *Thigh* corresponds to γ_8, which has the next highest posterior sampling probability of being nonzero after *Waist* and *Fore*. The model inferred from the graphical analysis therefore agrees with the model inferred from the *leaps* output.

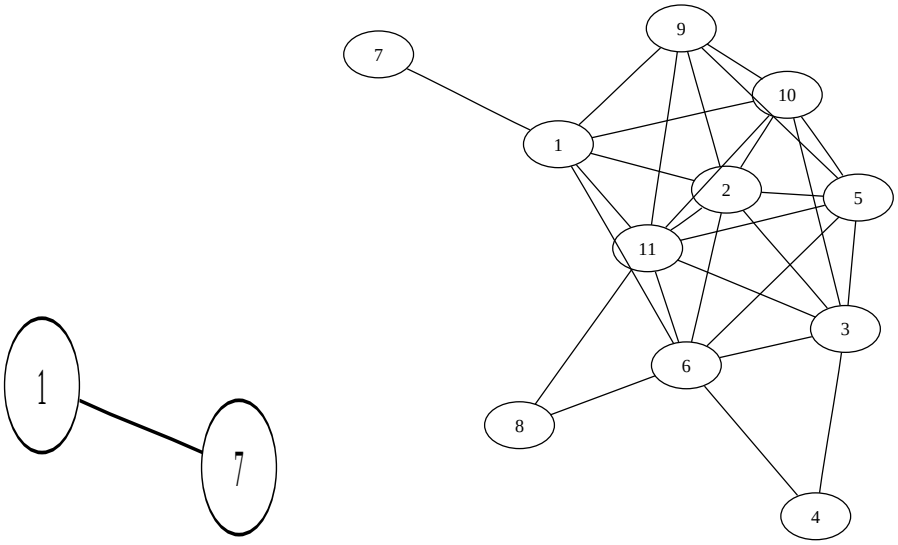
However, the graphical method gives insight into the interrelationships between all variables simultaneously, whilst the *leap* analysis does not. In particular, it is possible to analyse whether or not *any* pair of dependent variables remain dependent after conditioning on all, or some proper subset, of the remaining variables. This is done by seeing if the edge posterior probabilities in the estimate of $g(\Omega)$ suggest that the variables are adjacent, or merely connected. The model presented in Subsection 5.4.3 provides estimates of $g(\Omega)$ in the case where all 11 variables are included as 'responses'. These pictures can be used to analyse the strength of the various conditional independencies between any of the total 11 variables in the dataset. This is not possible using the *leap* analysis.

Figure 6.9 shows the 95% and 65% graphs respectively, for model $DCPCSVS$ using the equicorrelated form of Φ. Recall that the $p = 11$ variables are indexed in the following order:

1. Mass: weight in kg,
2. Fore: maximum circumference of forearm,
3. Bicep: maximum circumference of bicep,
4. Chest: distance around chest directly under the armpits,

5. Neck: distance around neck, approximately halfway up,
6. Shoulders: distance around shoulders, measured around the peak of the shoulder blades
7. Waist: distance around waist, approximately trouser line,
8. Height: from top of head to toe,
9. Calf: maximum circumference of calf,
10. Thigh: circumference of thigh, measured halfway between the knee and the top of the leg,
11. Head: maximum circumference of head.

Panel (a) suggests that *Mass* and *Waist* are most likely to be dependent, regardless of which of the remaining variables are conditioned upon. Panel (c) suggests that *Mass* may well be adjacent to *Fore*, as suggested by the summary analysis from Larner (1996). This model can be tested further by considering similar graphs at varying levels of posterior sampling probability between 65% and 95%. For brevity, this analysis is excluded, but the discussion illustrates that graphical techniques provide a method for deciding the structure of the linear regression model.



(a) (b)

Figure 6.9: Graphs for the physical measurements dataset where only the mean is included in the regression model. Panel (a) is 95% graph for model $DCPCSVS$ using the equicorrelated form of Φ. Panel (b) is the 65% graph for model $DCPCSVS$ using the equicorrelated form of Φ.

Chapter 7

Evaluating and assessing the size prior for a graph

7.1 Introduction

This chapter gives details on the size prior for a graph. This prior is introduced in Section 4.7. We begin by comparing its performance with the uniform prior, then present the methodology for calculating and estimating the $A_{p,k}$.

7.2 Comparison of the size prior for a graph with the uniform prior

This section compares the prior based on the graph size with the uniform prior that is used in most previous articles. Performance is in terms of a loss function and a simulation is carried out to numerically assess performance. We find that overall the size based prior for g outperforms the uniform prior.

Our simulation considers the following five graph types for g. (a) $\Omega = I$, the identity matrix, representing the empty graph and a diagonal covariance matrix; (b) Ω tridiagonal, representing a sparse and decomposable graph which is a path consisting of $p-1$ edges; (c) Ω an 'extreme' full matrix (the correlation coefficients ρ_{ij} of Ω^{-1} satisfy $|\rho_{ij}| > .30$), which corresponds to a complete graph; (d) Ω corresponding to a 4-cycle on p vertices representing a sparse but nondecomposable graph; and (e) Ω corresponding to a $p-$cycle on p vertices, again representing a sparse but nondecomposable graph. We note that

the nondecomposable graphs in (d) and (e) require the addition of extra edges when we estimate them by a mixture of decomposable graphs. Furthermore, (e) is an extreme case of non-decomposability, as it requires the addition of at least $p - 3$ edges to make the graph decomposable. Conversely, the unchorded 4-cycle on p vertices requires the addition of only one edge to make it decomposable, so is chosen as an indicator of performance for the sparsest nondecomposable case.

The simulation considers the three forms of Φ described in Section 4.6 and two sample sizes $n = 40$ and $n = 100$. We report results for matrices of size $p = 17$, but similar results are obtained for matrices of other sizes.

The design of the simulation study is similar to that in Section 4.14. We use L_1 as the loss function, as described in Section 4.14.

We use boxplots to compare replication by replication the size-based prior with the uniform prior in terms of the percentage *increase* in the loss function L_1 resulting from using the uniform prior compared to the size-based prior. That is, the boxplots are based on calculating

$$100(L_1^{unif} - L_1^{size})/L_1^{size}$$

for each replication, where L_1^{unif} and L_1^{size} are the values of $L_1(\widehat{\Sigma}, \Sigma_T)$ for the uniform and size-based priors respectively.

The boxplots are based on 20 replications with each replication consisting of 2,000 burn-in iterations and 20,000 sampling iterations. We ran the sampler for the case $p = 17$ on $n = 40$ and 100 observations from five simulated data sets corresponding to the five models (a)–(e) for Ω.

Figure 7.1 presents the results for $p = 17$. The plots show that for $\Phi = \tau I$ and Φ equicorrelated, the size prior is at least as good, and often much better than, the uniform prior. For $\Phi = \tau S_y/(n-1)$, the comparison between the size prior and the uniform prior is inconclusive for $n = 40$, but for $n = 100$ the size prior is at least as good as, and often better than the uniform prior. We conclude that the size-based prior outperforms the uniform prior.

We also compared the performance of the three forms of Φ for the uniform and size priors. Figures 7.2 and 7.3 present indicative results. We find that overall the equicorrelated form of Φ using the size-based prior for the graph performs best. Therefore it is this combination that we use for comparison with other covariance selection models.

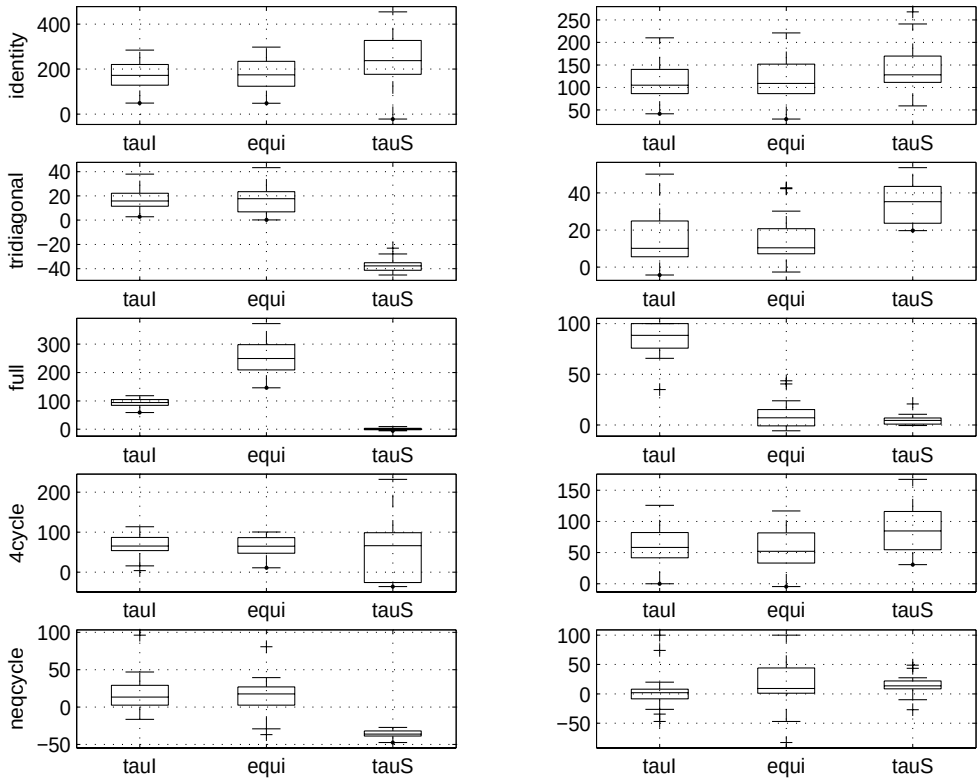


Figure 7.1: Percentage increase in loss of uniform prior relative to the size prior measured under L_1 loss. The left panels correspond to $n = 40$ and the right panels to $n = 100$. tauI, equi and tauS correspond to $\Phi = \tau I$, Φ equicorrelated and $\Phi = \tau S_y/(n-1)$.

7.3 Evaluating the size-based prior

To use the size-based prior for graphs on p vertices, we need the set of numbers $\{A_{p,k} : k = 0, \ldots, r\}$ where $A_{p,k}$ is the number of decomposable graphs of size k on p vertices, and $r = \binom{p}{2}$ is the maximum graph size. These numbers are not in the literature, nor is there a general method available for computing them. In this section we present some exact values of $A_{p,k}$ as well as a simulation method that can estimate the $A_{p,k}$ as precisely as necessary.

Let $B_{p,k}$ be the number of connected decomposable graphs of size k on p vertices. Equations (3) and (4) of Castelo & Wormald (2001) give recurrences to calculate $A_{p,k}$ from the $B_{p,k}$ analytically, and the information to calculate all $B_{p,k}$ analytically is implicit in Wormald (1985). For $p \leq 8$, Wormald (1985) gives the $B_{p,k}$ from which we computed the

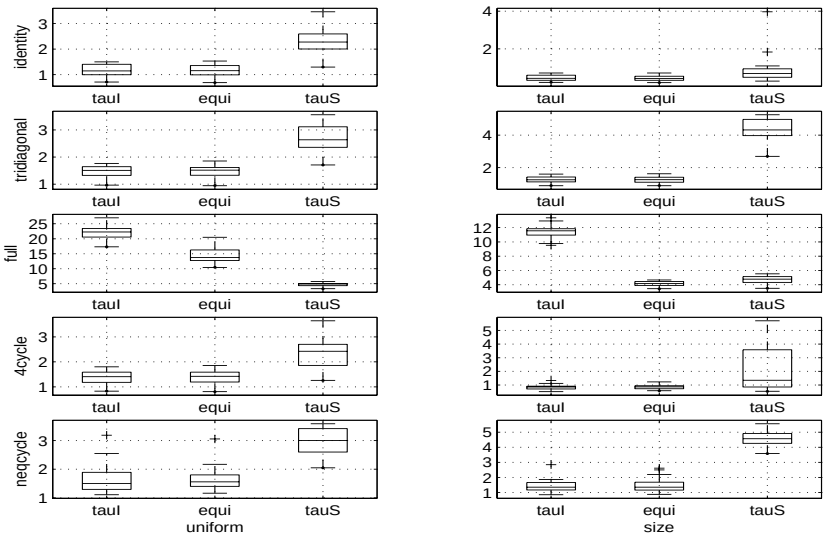


Figure 7.2: Values of L_1 for the three forms of Φ and the five different forms of Ω. The sample size is $n = 40$ with the left panel the uniform prior and the right panel the size-based prior.

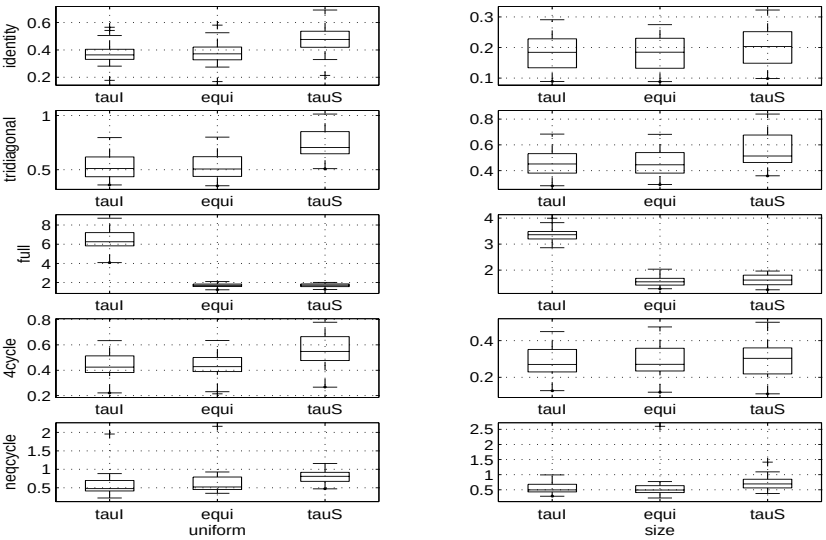


Figure 7.3: Values of L_1 for the three forms of Φ and the five different forms of Ω. The sample size is $n = 100$ with the left panel the uniform prior and the right panel the size-based prior.

$A_{p,k}$ and these are reported in Table 7.1.

However, Wormald's (1985) analytic approach for obtaining the $B_{p,k}$ is likely to be computationally intractable for $p > 25$ (private correspondence with Wormald). Even for $8 < p \leq 25$ obtaining the $B_{p,k}$ would take weeks on realistically sized computers.

Table 7.1: For each p, $2 \leq p \leq 8$ the table gives each $A_{p,k}$, $0 \leq k \leq r$ and $A_p = \sum_{k=0}^{r} A_{p,k}$. The table also gives for each p the percentage of graphs that are decomposable.

k	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1
1	1	3	6	10	15	21	28
2		3	15	45	105	210	378
3		1	20	120	455	1330	3276
4			12	195	1320	5880	20265
5			6	180	2526	18522	92988
6			1	140	3085	40647	315574
7				90	3255	60795	770064
8				30	3000	79170	1357818
9				10	2235	92785	2078300
10				1	1206	94521	2892176
11					615	81417	3621576
12					260	58485	4016439
13					60	40110	3916724
14					15	24255	3432660
15					1	12222	2855748
16						4872	2185484
17						1890	1488984
18						595	902944
19						105	493220
20						21	258468
21						1	118504
22							46046
23							14868
24							4690
25							1176
26							168
27							28
28							1
$\sum_{k=0}^{r} A_{p,k}$	2	8	61	822	18,154	617,675	30,888,596
% decomposable	100%	100%	95%	80%	55%	29%	12%

Furthermore, analytically deriving the $A_{p,k}$ from the $B_{p,k}$ is computationally feasible only for small p. Because of these difficulties we propose a simulation methodology to estimate the $A_{p,k}$ for all p.

7.4 Simulation methodology for estimating the $A_{p,k}$.

We begin with some exact results which can be used to calculate $\{A_{p,k} : k \leq 5 \text{ and } r-2 \leq k \leq r\}$ analytically for any p. Let $F_{p,k}$ denote the number of nondecomposable graphs having p vertices and k edges.

Lemma 7.4.1 *1.* $A_{p,k} = \binom{r}{k} - F_{p,k}$.

2. $F_{p,0} = F_{p,1} = F_{p,r} = 0,\ p \geq 0$.

3. $F_{p,2} = F_{p,r-1} = 0,\ p \geq 2$.

4. $F_{p,3} = 0,\ p \geq 3$.

Proof. The proof is obvious. ■

Lemma 7.4.2 *1. For* $p \geq 4$, $F_{p,4} = \binom{p}{4} \times 3$.

2. For $p \geq 4$, $F_{p,r-2} = F_{p,4}$.

3. For $p \geq 5$, $F_{p,5} = \binom{p}{5} \times 12 + \binom{p}{4} \times 3 \times (r-6)$.

Proof. See Wong (2002) or Appendix 8.3. ■

We now show how to estimate the $\{A_{p,k} : 6 \leq k \leq r-3\}$ for all p. Our approach is to run a separate simulation to estimate each $A_{p,k}$ for $6 \leq k \leq r-3$. The simulations are done in ascending order of k, i.e. $k = 6, \ldots, r-3$, and the simulation to estimate a particular $A_{p,k}$ is restricted to graphs of size $\leq k$ and uses the estimates $\widehat{A}_{p,j}$ of $A_{p,j}$ for $j = 6, \ldots, k-1$ that have been calculated in previous simulations.

We now describe the details of the simulation to estimate a particular $A_{p,k}$. Let $\phi_{p,k}$ be the initial estimate of $A_{p,k}$ given by

$$\phi_{p,k} = \widetilde{\alpha}_{p,k} \frac{\widehat{A}^2_{p,k-1}}{\widehat{A}_{p,k-2}} \tag{7.1}$$

with $\widetilde{\alpha}_{p,k}$ chosen in the range $(0.5, 1)$. To justify this choice of $\phi_{p,k}$, we note that we have found empirically that $\log A_{p,k}$ is approximately a negative quadratic (see figures 7.4 and

7.5) so that $\log A_{p,k} - 2\log A_{p,k-1} + \log A_{p,k-2} \leq 0$, and hence

$$\alpha_{p,k} = \frac{A_{p,k}/A_{p,k-1}}{A_{p,k-1}/A_{p,k-2}} \leq 1.$$

We have also found empirically that $\alpha_{p,k}$ is likely to exceed 0.5. Because

$$A_{p,k} = \alpha_{p,k}\frac{A_{p,k-1}^2}{A_{p,k-2}}$$

the above discussion suggests the choice of $\phi_{p,k}$ in (7.1).

We use Lemmas 7.4.1 and 7.4.2, the estimates $\widehat{A}_{p,j}$ of $A_{p,j}$ for $j = 6, \ldots, k-1$ that have been calculated in previous simulations, and the initial estimate $\phi_{p,k}$ of $A_{p,k}$ given above to define the following probability distribution $p_e(g)$ on the graphs g of size $\leq k$. To simplify the notation we omit subscripts for p and k in $p_e(g)$. Let

$$p_e(g) \propto \begin{cases} \frac{1}{A_{p,size(g)}} & \text{if } 0 \leq \text{size}(g) \leq 5 \\ \frac{1}{\widehat{A}_{p,size(g)}} & \text{if } 6 \leq \text{size}(g) \leq k-1 \\ \frac{1}{\phi_{p,k}} & \text{if size}(g) = k \end{cases} \tag{7.2}$$

which implies that

$$\begin{aligned} \frac{p_e(size = k)}{p_e(size \leq 5)} &= \frac{A_{p,k}/\phi_{p,k}}{\sum_{j=0}^{5} A_{p,j}/A_{p,j}} \\ &= \frac{1}{6}A_{p,k}/\phi_{p,k} \end{aligned}$$

and hence

$$A_{p,k} = 6\phi_{p,k}\frac{p_e(size = k)}{p_e(size \leq 5)}.$$

By running the simulation described below based on $p_e(g)$ we can estimate the ratios $p_e(size = k)/p_e(size \leq 5)$ by their relative frequencies and hence obtain an estimate of

$$\widehat{A}_{p,k} = 6\phi_{p,k}\frac{\hat{p}_e(size = k)}{\hat{p}_e(size \leq 5)},$$

where $\hat{p}_e(size = k)$ and $\hat{p}_e(size \leq 5)$ are the empirical relative frequencies.

The simulation uses the following MCMC sampling scheme. As in Section 4.9, we generate the edge indicators one at a time conditional on the other edge indicators. Let $g^c = (V, E^c)$ be the current graph with edge indicators given by $\{e_{kl} : (k,l) \in E^c\}$. We select an edge (i,j) at random. If $g = (e_{ij}, e^c_{-ij})$ corresponds to a decomposable graph of size $\leq k$ for both $e_{ij} = 0$ and $e_{ij} = 1$ then we proceed, where we again use the legal

edge addition and deletion characterizations of Giudici & Green (1999) and Frydenberg & Lauritzen (1989) respectively to test this. Otherwise we select a new edge. If we proceed, then we propose a new graph $g^p = (1 - e^c_{ij}, e^c_{-ij})$ and accept this graph with probability

$$\min\{1, p_e(g^p)/p_e(g^c)\}$$

which is evaluated using (7.2).

We note that at each stage we can also re-estimate $A_{p,j},\ j = 6, \ldots, k-1$.

7.5 Results

This section presents the estimates $\widehat{A}_{p,k}$ for $k = 0, \ldots, r$ and $p = 8$ and 34, and provides a general method to check on the quality of these estimates. Define the prior $p_e(g)$ on the decomposable graphs g as

$$p_e(g) \propto \begin{cases} \frac{1}{A_{p,size(g)}} & \text{if } 0 \leq \text{size}(g) \leq 5 \text{ or } r-2 \leq \text{size}(g) \leq r \\ \frac{1}{\widehat{A}_{p,size(g)}} & \text{if } 6 \leq \text{size}(g) \leq r-3. \end{cases}$$

The prior p_e in this section is different to p_e in Section 7.4. If the estimates $\widehat{A}_{p,k}$, $6 \leq k \leq r-3$ are precise, then $p_e(size = k)$ should be close to uniform and hence close to the target value $1/(r+1)$. An approximate lower bound for the standard error of the estimates of $p_e(size = k)$ is $\sqrt{\pi(1-\pi)/J}$, where $\pi = 1/(r+1)$ and J is the number of iterates used to compute $p_e(size = k)$. Our simulations use a burn-in period of 2,000 iterations and a sampling period of $N = 10,000$ iterations. Figure 7.4 plots the estimates $\widehat{A}_{p,k}$ for $p = 8$ and the true values $A_{8,k}$, $k = 0 \ldots r$ on both an absolute and logarithmic scale. Figure 7.4 also plots the estimates of $p_e(size = k)$ together with the target value $1/(r+1)$ and lower bounds for the ± 3 standard error lines.

Figure 7.5 has the same interpretation as Figure 7.4 but is for $p = 34$. The true values of $A_{34,k}$ are not plotted as they are mostly unknown. Similar plots were obtained for $9 \leq p \leq 40$, but these are omitted for brevity.

We now describe how to check the quality of these estimates. If $\widehat{A}_{p,j} = A_{p,j}$ for all j then $\widehat{p}_e(size = j) = 1/(r+1)$ for all j so that $E(\widehat{p}_e(size = j)) = 1/(r+1)$ and the standard error of $\widehat{p}_e(size = j)$ is at least as large as $\sqrt{\pi(1-\pi)/N}$ where $\pi = 1/(r+1)$ and N is the number of iterates used to calculate $\widehat{p}_e(size = k)$. To account for autocorrelations in the MCMC scheme we use every 20^{th} iterate. Figure 7.4(a) and Figure 7.5(a) give plots of the $\widehat{A}_{p,j}$ for $p = 8$ and 34, together with their respective logarithmic scale values, $\widehat{p}_e(size = k)$ and the horizontal lines at $1/(r+1)$ and $1/(r+1) \pm 3\sqrt{\pi(1-\pi)/J}$ in panel (a).

Figure 7.4(a) also shows the true values $A_{8,j}$. Similar plots of true compared to estimated values were achieved for $p \leq 7$.

For $p = 9, \ldots, 12$ the totals $A_p = \sum_j A_{p,j}$ are known, but not the values $A_{p,j}$. As a further check on results we compared our estimated values of $\widehat{A}_p$ to A_p and found that we were consistently within 1% of the truth.

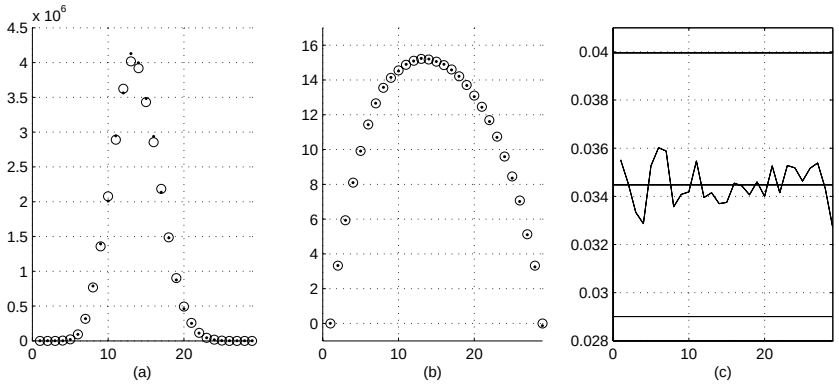


Figure 7.4: Panel (a): Plot of true $A_{8,k}(\cdot)$ and estimates $\widehat{A}_{8,k}$ (open circles), $k = 0, \ldots r$. Panel (b): Log scale of plot (a). Panel (c): Plot of $\hat{p}_e(size = k)$ together with their target value of $1/(r+1)$ (middle horizontal line) and ± 3 approximate standard errors (outer horizontal lines).

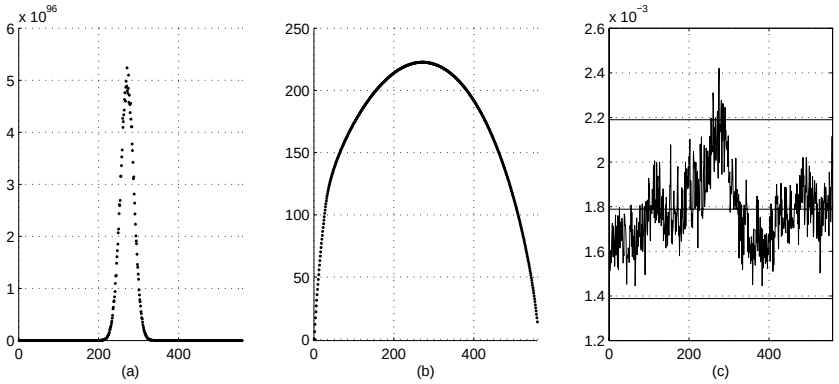


Figure 7.5: Panel (a): Plot of estimates $\widehat{A}_{34,k}$ $k = 0, \ldots r$. Panel (b): Log scale of plot (a). Panel (c): Plot of $\hat{p}_e(size = k)$ together with their target value of $1/(r+1)$ (middle horizontal line) and ± 3 approximate standard errors (outer horizontal lines).

Chapter 8

Appendicies

8.1 Appendix A: MATLAB code with line by line explanations for the reduced conditional sampler and covariance selection using the methodologies presented in this thesis.

This section gives line by line descriptions of working MATLAB code for achieving each of the corresponding subsections of Section 3.5. The equivalent FORTRAN code is given, without explanations, in Appendix 8.4

8.1.1 checking chordality

Subsections 3.5.2 and 3.5.3 illustrate that a graph is chordal if the numbering $v_1, \ldots, v_p$ is such that each v_i has a maximal number of previously numbered neighbours, $nbrs(v_i) \cap \{v_1, \ldots, v_{i-1}\}$, and that this set is complete. Define $pa(v_i) = nbrs(v_i) \cap \{v_1, \ldots, v_{i-1}\}$ as the *parents* of v_i. The Maximum Cardinality Search of Tarjan & Yannakakis (1984) explained below starts with a set of unnumbered vertices, and sequentially numbers vertices by maximising $pa(v_i)$. As each vertex is numbered, it checks that $pa(v_i)$ is complete and aborts when this condition is not satisfied, returning 'not chordal'. If the algorithm succeeds in numbering all vertices, then the graph is complete and the sequence of vertices form a perfect numbering.

The following 22 item list description is enumerated with respect to the 22 lines of code (excluding comment and blank lines) that follows it.

1. set all diagonals of g to 1 so that completeness can be checked by comparing to a string of ones.

2. define p as the number of vertices.

3. initialise *order* as a $1 \times p$ vector of zeros. *order* is the $1 \times p$ permutation vector of indicies $1, \ldots p$ ($1, \ldots p$ is the original indexing in the adjacency matrix g) that gives a perfect numbering.

4. initialise *chordal* to 1 (yes).

5. arbitrarily set v_1 as the first in the perfect numbering; i.e. initialise the first element of the vector *numbered* to 1.

6. arbitrarily set v_1 as the first in the perfect numbering; i.e. set $order(1) = 1$.

7. begin for loop $i = 2, \ldots, p$.

8. for each i, form the set *capU* of unnumbered vertices.

9. for each i, initialise *score* to the zero vector. In what follows $score(i)$ will equal $|p(v_i)|$.

10. for each i, begin for loop $u_i = 1, \ldots, length(capU)$, ($length(capU)$ is the number of currently unnumbered vertices).

11. initialise u as the u_ith element of *capU*.

12. set $score(u_i)$ as the number of previously numbered neighbours ($length(intersect(neighbours_node(g, u), num$ is the number of the intersection between the neighbours of u and *numbered*, the vector of numbered vertices)

13. end inner u_i loop

14. find the position of the maximum of score ($argmax(score)$), and set u equal to *capU* at this position. Thus u is the first element of the vector of unnumbered vertices which has the maximal number of parents.

15. add u to the set of numbered vertices.

16. set the ith element of *order* equal to u, the perfect numbering permutation of the vertices.

17. find pa, the set of already numbered neighbours, or parents, of u.

18. begin if loop to test noncompleteness of pa by seeing if there is at least one position of g corresponding to one pair of vertices in pa that are not joined by an edge. That is, the submatrix $g(pa, pa)$ is not all ones (assuming extra edges $g(v, v)$ are added).

19. if the submatrix $g(pa, pa)$ is not complete, set chordal=0 (no).

20. if the submatrix $g(pa, pa)$ is not complete, break out of the external i loop and abort.

21. end internal ith test for completeness

22. end external $for\ i = 1, \ldots, p$ loop

```
function [chordal, order]=check_chordal(g)
% inputs: 1. g, the p x p symmetric adjacency matrix with
%               respect to an original ordering v_1, ..., v_p

% output: 1. chordal=1/0 (yes/no)
%         2. order=[a permutation vector of the v_i].
             order is a sequence alpha(1)...alpha(p) which is an ordering
             of the v_i such that the sequence v_alpha(1)...v_alpha(p)
             is a perfect numbering if it exists]

% A numbering alpha is perfect if
% nbrs(alpha(i)) intersect {alpha(1)...alpha(i-1)} is complete.
% A graph is chordal iff it has a perfect numbering.
% The Maximum Cardinality Search algorithm will create such a
% perfect numbering if possible.
% See Golumbic, "Algorithmic Graph Theory and Perfect Graphs",
% Cambridge Univ. Press, 1985, p85.
% or Castillo, Gutierrez and Hadi,
% "Expert systems and probabilistic network models", Springer 1997, p134.

   g=setdiag(g, 1);
   p = size(g,1);
   order = zeros(1,p);
   chordal = 1;
```

```
numbered = [1];
order(1) = 1;
for i=2:p
   capU = setdiff(1:p, numbered);
      % unnumbered verticies
   score = zeros(1, length(capU));
   for u_i=1:length(capU)
     u = capU(u_i);
     score(u_i) = length(intersect(neighbours_vertex_cell(g, u), numbered));
   end
   u = capU(argmax(score));
   numbered = [numbered u];
   order(i) = u;
   pa = intersect(neighbours_vertex_cell(g,u), order(1:i-1));
            % already numbered neighbours
   if ~isequal(g(pa,pa), ones(length(pa)))
      chordal = 0;
      break;
   end
end
```

8.1.2 finding cliques, given the order

Subsection 3.5.2 illustrated how to find the cliques of a chordal graph. The process was as follows. Assume that $v_{alpha(1)}, \ldots, v_{alpha(p)}$ is a perfect numbering, It was pointed out in Subsection 3.5.2 that the set $v_{alpha(i)} \cup pa(v_{alpha(i)})$ of a vertex with its parents was either: (1) a strict subset of a single clique for vertices which were not the last ordered in a clique, (2) the whole of a single clique for vertices which were the last ordered in a clique.

Therefore, the sequence of numbers of parents of the variables will be constant or decreasing whenever the vertex is the last to be ordered in a clique. This is the principle employed to find the cliques algorithmically as follows.

Create a vector num_{pa} of prenumbered neighbours such that *num_pa(i)* is the number of parents of the ith variable $v_{order(i)}$. If *num_pa* is decreasing (*num_pa* i $\geq$ *num_pa(i+1)*, then the set comprising the vertex $v_{order(i)}$ and its parents is the next clique in the perfect sequence. Note that whenever *num_pa* is constant, all the associated vertices have the same maximal number of parents and any permutation of the associated subsequence of cliques

within the overall ordering of all cliques, will still result in a perfect sequence ordering of the cliques.

The following 33 item list description is enumerated with respect to the 33 lines of code (excluding comment and blank lines) that follows it. The code is first explained assuming a MATLAB *cell array* representation of the cliques, followed by the equivalent explanation for a MATLAB matrix array representation.

1. define $p =$ number of vertices in g.
2. initialise a $1 \times p$ cell array pa of p empty sets for the parents of each vertex. The sets of parents will be used to create the cliques, so each set pai is saved in pa rather than recomputed later.
3. intialise to zero the vector *num_pa* for recording the number of parents.
4. begin for loop $i = 2, \ldots p$ to find sequentially the number of parents of each vertex, where the sequence ordering is the same as the maximum cardinality search perfect numbering.
5. set v as the ith vertex in $order$, the maximum cardinality search perfect numbering.
6. create *pre_v*, the set of all predecessors of v with respect to the perfect numbering.
7. find the set ns of neighbours of v in g.
8. find the set of parents pai by taking the intersection of the neighbours with the predecessors.
9. record the number of parents of the ith vertex $v = v_{order(i)}$ in *num_pa*. Note that the ith element of *num_pa* corresponds to the number of parents of the ith node with respect to the permutation vector order; i.e. $v = v_{order(i)}$, and NOT $v = v_i$. It is critical that the variables retain the order as per the maximum cardinality search perfect numbering given by the input permutation vector *order*.
10. end for loop $i = 2, \ldots p$.
11. initialise $ladder$ to zero.
12. begin for loop $i = 1, \ldots p$ to record which vertices are 'ladder' vertices; i.e. the vertices v such that $v \cup pa(v)$ is a clique.
13. begin if test for decreasing number of parents.

14. if the number of parents of $v = order(i)$ decreases as the next $i + 1$, or the vertex $v = v_{order(p)}$ is the last to be considered, then make that vertex the ith ladder vertex $ladder(i)$.

15. end internal test for decreasing number of parents.

16. end external for loop $i = 1, cdotsp$.

17. initialise a cell array for the cliques of the same size as $ladder$, so that the maximum number of cliques possible is assumed.

18. begin for loop $i = 1, \ldots p$ to create each clique, assuming that the number of cliques is p; i.e. the maximum possible. number

19. begin if test for the nonzero elements of $ladder$.

20. if the ith element of ladder is still zero (recall ladder was initialised as a zero vector), then there is no associated clique. i.e. the vertex $v_{order(i)}$ is not the last numbered in the clique. So set *cliquesi* equal to the empty set, [].

21. otherwise,

22. set *cliquesi* as the union of $v_{order(i)}$ and its parents. Note that MATLAB orders union(a,b) as [min(a,b), max(a,b)] regardless of the relative sizes of a, b

23. end if test for finding the ladder vertices.

24. end for loop for clique creation.

25. now we need to get rid of the empty cliques, so define k as the number of nonzero elements in ladder.

26. initialise a counter variable new_index to 1.

27. initialise *re_index_cliques* as a $1 \times k$ cell array of k empty sets.

28. begin for loop $i = 1, \ldots p$ to find the nonempty elements of *cliques*.

29. begin if test for finding the nonzero ladder vertices.

30. if $ladder(i)$ is nonzero, then define the *new_index*th set of *re_index_cliques* to be *cliquesi*.

31. add one to *new_index* so that it can be used to index the next nonempty clique in *cliques*.

32. end if test for finding the nonzero ladder vertices.

33. end for loop for finding the set *re_index_cliques* of nonempty cliques.

```
function [re_index_cliques, cliques]=chordal_to_ripcliques_cell(g, order)
% inputs: 1. g, the p x p symmetric adjacency matrix with
%            respect to an original ordering v_1, ..., v_p
%              2. order, the output of check_chordal or any other
%            perfect numbering permutation of the vertex indicies.

% output: 1. re_index_cliques, a cell representation containing only
%            non-empty cliques, re_indexed
%            so that re_index_cliques{i} is the ith non-empty clique
%            in a sequence of cliques that satisfies the
%            running intersection property.
%         2. cliques, a cell representation of the
%            cliques of g in perfect order.
%            i.e. Unlike re_index_cliques, these
%            are indexed with respect to the vertices in the ordering.
%            So if order(j) is a ladder vertex, the cell will contain the associated clique.
%            If not, the cell will be empty.
%
% Note:   If cliques{i}={2, 4, 5, 7}, then clique{i} comprises
%         variables v_2, v_4, v_5 and v_7 with respect to the original
%         ordering of the adjacency matrix, and not the perfect numbering.

  p=size(g,1);
  pa=cell(1,p);
  num_pa=zeros(1,p);
            %initialise the vector of number of predecessors

  for i=2:p;
      v=order(i);
      pre_v=order(1:i-1);
      ns=neighbours_vertex_cell(g, v);
            % find set of neighbours of each v=order(i)
            % turn the set into a vector (so can take intersection)

      pa{i}=intersect(ns, pre_v);
                        % find the sets of those neighbours which precede
                        % v(i) with respect to order.
                        % Store answer for cliques.

      num_pa(i)=length(pa{i});
          % get cardinality for ladder test. note that the ith element of num_pa
          % corresponds to the number of pre-nbs of the ith vertex in order; i.e
          % v=order(i), and NOT v=i. We need to keep variables ordered as per the
```

```
        % mcs ordering, the vector called order.
  end;

  ladder=zeros(1,p);

  for i=1:p;
    if i==p | num_pa(i) >= num_pa(i+1);
                  %if i=p or cardinality of pa decreasing with i
                  %then the vertex v=order(i) is a ladder vertex.
      ladder(i)=order(i); %make this v the next ladder vertex
    end;
  end;

  cliques=cell(size(ladder));
  for i=1:p;
    if ladder(i)==0
      cliques{i}=[];
    else;
       cliques{i}=union(order(i), pa{i});
         % NOTE matlab orders union(a,b) as [min(a,b), max(a,b)]
         % regardless of relative size of a,b
    end;
  end;

% get rid of empty cliques

  k=length(find(ladder));
  new_index=1;
  re_index_cliques=cell(1,k);
  for i=1:p;
     if ladder(i)~=0 re_index_cliques{new_index}=cliques{i};
       new_index=1+new_index;
     end;
  end;

% Theorem: re_index_cliques{i} are the cliques of g and clique ordering satisfies the RIP.
```

The code is now explained assuming a MATLAB matrix array representation of the sets of variables. In this representation, a subset of p variables is represented by a $p \times 1$ column vector of zeros and ones. If there is a one in the ith row position, then the vertex variable v_i is an element of the subset. A zero indicates v_i is not an element. A matrix array comprised of each of these columns is then used to represent a set of subsets. If there is a one in the i, jth position of the array, then the ith variable is an element of the jth subset. Since the smallest nonempty clique subset consists of a single variable, the largest number of cliques possible for a given graph is p. Thus a $p \times p$ array *cliques* is sufficient for representing any set of cliques. For example, if $V = \{1, 2, \ldots, 8\}$ and clique $C_2 = \{2, 4, 5\}$,

then *cliques(:,2)*= [01011000]′. If there are $k < p$ cliques, then *cliques(:, k+1:p)*= [0].

The following 28 item list description is enumerated with respect to the 28 lines of code (excluding comment and blank lines) that follows it. Note that the subscript _zo used throughout, indicates a 'zero / one' representation of the variables.

1. define $p =$ number of vertices in g.
2. initialise a $p \times p$ matrix array *pa* to zero for the parents of each vertex. The sets of parents will be used to create the cliques, so each set *pa(:,i)* is saved rather than recomputed later.
3. intialise to zero the vector *num_pa* for recording the number of parents.
4. begin for loop $i = 2, \ldots p$ to find sequentially the number of parents of each vertex, where the sequence ordering is the same as the maximum cardinality search perfect numbering.
5. set v as the ith vertex in $order$, the maximum cardinality search perfect numbering.
6. initialise to zero the vector *pre_v* to represent the set of all predecessors of v with respect to the perfect numbering.
7. set *pre_v(j)*= $1, j = 1, \ldots, i-1$ equal to one to represent that variables $v_1, \ldots v_{i-1}$ are the predecessors of v_i with respect to the perfect numbering.
8. find the set ns of neighbours of v in g.
9. find *pa_i*, the column vector representing the set of parents of v in g, by taking the intersection of the neighbours of v with its predecessors.
10. to find the number of parents for *num_pa*, find the nonzero entries of *pa_i*. Set *num_pa(i)* equal to the number of parents of the ith vertex $v = order(i)$. Note that *num_pa(i)* corresponds to the number of parents of the ith node in *order*; i.e. $v = order(i)$, and NOT $v = i$. It is critical that the variables retain the order as per the maximum cardinality search perfect numbering given by the permutation vector $order$.
11. Save the set of parents in *pa*; i.e. set *pa(:,v)=pa_i*.
12. end for loop $i = 2, \ldots p$.
13. initialise $ladder$ to zero.

14. begin for loop $i = 1, \ldots p$ to record which vertices are 'ladder' vertices; i.e. the vertices v such that $v \cup pa(v)$ is a clique.

15. mtest for decreasing number of parents. For the test, the number of parents of the ith vertex according to *order* is found by summing the *order(i)*th column of *pa*.

16. if the number of parents of $v = order(i)$ decreases for the next $i + 1$, or the vertex $v = v_{order(p)}$ is the last to be considered, then make that vertex the ith ladder vertex $ladder(i)$.

17. end internal test for decreasing number of parents.

18. end external for loop $i = 1, \ldots p$.

19. initialise a $p \times p$ matrix array for the cliques.

20. intialise to zero *index_non_zero_ladder*, a counter used to index the nonzero elements of *ladder*. Note that this counter will ensure all the leading columns are nonzero if there are fewer than p cliques.

21. begin for loop $i = 1, \ldots p$ to create each clique, allowing for the maximum number p possible cliques.

22. begin if test for finding the ladder vertices; i.e. represented by the nonzero elements of $ladder$.

23. initialise to zero *v_i*, a $p \times p$ column vector representation of the single variable $v = ladder(i)$.

24. to represent the single variable $v = ladder(i)$, the *ladder(i)*th variable of the column vector *v_i* must be equal to one, and every other entry must be zero.

25. add one to the counter *index_non_zero_ladder*.

26. set the *index_non_zero_ladder*th column of the array *cliques* as the union of v with its parents.

27. end if test for finding the ladder vertices.

28. end for loop for clique creation.

```
function [cliques]=chordal_to_ripcliques_zo(g, order)
% inputs: 1. g, the p x p symmetric adjacency matrix with
%            respect to an original ordering v_1, ..., v_p
%        2. order, the output of check_chordal or any other
%            perfect numbering of the vertex indicies.

% output: 1. cliques, a p x p matrix representation of the
%            cliques of g in perfect order.
%            i.e. each column cliques(:,i) is the ith ordered clique
%            in a sequence of cliques that satisfies the running intersection property.
%            If the jth row of cliques(:,i) equals one, then v_j is an
%            element of clique C_i.

%            Note: the maximum number of cliques for any graph is p,
%            which occurs when the v_i are all independent so each clique
%            comprises a single variable vertex.

   p=size(g,1);
   pa=zeros(p,p);
   num_pa=zeros(1,p);
            % initialise the vector of number of predecessors

   for i=2:p;
      v=order(i);
      pre_v=zeros(p,1);
      pre_v(order(1:i-1))=1;
          % create 0/1 col vector representing predecessors of v
      ns=neighbours_vertex_zo(g, v);
          % find set of neighbours of each v=order(i)
      pa_i=intersect_zo(ns, pre_v);
      num_pa(i)=size(find(pa_i),1);
      pa(:, v)=pa_i;
          % find the sets of those neighbours which precede
          % v=order(i) with respect to order. Store answer for cliques.
          % so if order=[1 3 7 5 2 4 6], pa(:,order(4)=5)=[0 0 1 0 0 0 1]'
          % pa =
              %      0     0     1     0     0     0     1
              %      0     0     0     1     0     0     0
              %      0     0     0     0     1     0     1
              %      0     0     0     0     0     0     0
              %      0     0     0     0     0     0     0
              %      0     0     0     0     0     0     0
              %      0     1     0     0     1     1     0
          % num_pa ordered as per order (i)  = 0 1 2 2 1 1 1
          % corresponding to v= 1 3 7 5 2 4 6

          % eg num of pre nbs of 3 for ladder test=sum(:,3), etc.
          % note that the ith column
          % corresponds to the pre-nbs of the ith vertex in order;
          % i.e v=order(i), and NOT v=i. Simly for cardinality
```

```
         % Need to keep variables ordered as per the
         % mcs ordering, the vector called order.
   end;

   ladder=zeros(1,p);
         % NOTE: for ladder(i)~=0, ladder(i)=order(i)
   for i=1:p;
      if i==p | sum(pa (:, order(i))) >= sum(pa(:,order(i+1)))
            %if i=p or cardinality of pa decreasing with
            % v=order(i), then the vertex v=order(i) is a ladder vertex.
            % eg, i=4, order as above has ladder [0 0 7 5 2 4 6]
            % as == or decrease occurs num_pa(i), i=3 4 5 6 7
            % and order(3)=7,order(4)=5,order(5)=2,order(6)=4,order(7)=6.
         ladder(i)=order(i);
            % make this v the next ladder vertex
       end;
   end;

%%%%%%%%%%%%%%%%%%%%%%
%% NOTE: i output non-empty columns first, so clique ordering follows
%% matrix cliques column ordering.
%% if only 3 cliques, then cliques(:, 3:n)=zero(n,1);
%% cliques(:,i)=union of the ith non-zero ladder vertex and its predecessors)
%% eg cliques(:,3)=[0 1 0 0 0 0 0]+[0 0 0 0 0 0 1] = {2} U {7}
%% NOTE: for ladder(i)~=0, ladder(i)=order(i)

   cliques=zeros(p, p);
        %  only create clique columns if it's non-empty set
   index_non_zero_ladder=0;

   for i=1:p;
      if ladder(i)~=0;
         v_col_i=zeros(p,1);
                % create column vector rep. of v=ladder(i)
         v_col_i(ladder(i),1)=1;
                % ladder(i)=order(i)
         index_non_zero_ladder=index_non_zero_ladder+1;
         cliques(:, index_non_zero_ladder)=union_zo(v_col_i,pa(:,ladder(i)));
      end;
   end;

% Theorem: cliques(index_non_zero_ladder, 1:size_cliques) are the cliques of g
% and clique ordering satisfies the RIP.
```

8.1.3 creating the junction tree, given a perfect sequence of cliques

This code creates a junction tree from a perfect sequence of cliques based on the principle of maximising the size of the intersection of clique C_i with its predecessors in the sequence, as explained in Sections 3.4 and 3.5. The cell array version is described before giving an alternative matrix array version.

The following 12 item list description is enumerated with respect to the 12 lines of MATLAB code (excluding comment and blank lines) that follows it.

1. find t, the number of nonempty cliques of g.
2. initialise a $1 \times t$ vector *score* to record the size of the intersections $C_i \cap C_j$.
3. initialise *jt*, the adjacency matrix of the junction tree, allowing for the maximum number of clique vertices.
4. begin for loop $i = 2, \ldots t$ to find the (C_i, C_j) edge.
5. begin internal k dependent for loop $k = 2, \ldots i-1$ to find the sizes of the intersections $C_i \cap C_k, k < i$.
6. set *score(k)* equal to $|C_i \cap C_k|, k < i$.
7. end internal k dependent for loop.
8. begin if test for empty intersections, so that an edge is not added if all intersections $C_i \cap C_k, k < i$ are empty.
9. if the maximum of score is greater than zero, then find j, the position of the maximum. Note that MATLAB uses the first maximal element in the case of nonuniqueness.
10. add the edge (C_i, C_j) to the junction tree by setting the i, jth and j, ith entries of the adjacency matrix *jt* equal to one.
11. end the internal if test.
12. end outer i dependent for loop.

```
function [jtree]=ripcliques_to_jtree_cell(re_index_cliques)
% input:  re_index_cliques, a 1 x t cell array of nonempty cliques
%         of a chordal graph in RIP ordering
%         (such as re_index_cliques from chordal_to_ripcliques_cell.m).

% output: jtree, the adjacency matrix of a junction tree with respect
%         to cliques (not necessarily unique).

% WARNING: this algorithm only works for cliques outputted
% with respect to a maximum
% cardinality search ordering of the cliques, as then the cliques are
% already ordered with respect to the rank of the highest vertex in the
% clique. For example, if mcs ordering of vertices is [6 5 3 1 4 2], then a
% clique comprised of {6,1} will precede a clique comprised of {4,2}
% since 6 and 1 precede 4 and 2 in the ordering [6 5 3 1 4 2].

   t=size(re_index_cliques,2);
   score=zeros(1,t);
   jtree=zeros(t,t);

   for i=2:t;
      for k=1:i-1;
        score(k)=length(intersect(re_index_cliques{i}, re_index_cliques{k}));
      end;

      if max(score)~=0 ;
        % only add the edge if clique i IS connected to one of its
        % predecessors. if score is all zeros, then clique has no intersection
        % with any of its predecessors. Since the cliques are in RIP, it must
        % follow that we are no longer in the same connected component of
        % the graph.
        % if clique i has no intersection with any of the preceding cliques,
        % then the graph is disconnected, so the adjacency matrix will have
        % a zero row/column for this i, and we have a forest, not a j_tree.

        j=argmax(score);
        jtree(i,j)=1; jtree(j,i)=1;
       end;
   end;
```

The matrix array version that follows the description below is identical, except that the equivalent matrix array operations are used.

1. find p, the maximum possible number of cliques of g.

2. initialise to zeros a $1 \times p$ vector *clique_sizes* to record the number of variables in each clique. This can be computed by summing each column of *cliques*, since $cliques(i, j) = 1$ if and only if $v_i \in C_j$.

3. calculate the column totals and store in *clique_sizes*.

4. calculate *num_cliques*, the number of nonempty cliques by finding how many of the p entries in *clique_sizes* are nonzero.

5. initialise a $1 \times p$ vector *score* to record the size of the intersections $C_i \cap C_j$.

6. initialise *jt*, the adjacency matrix of the junction tree, allowing for the maximum number of clique vertices.

7. begin for loop $i = 2, \ldots,$ *num_cliques* to find the (C_i, C_j) edge.

8. begin internal k dependent for loop $k = 2, \ldots i-1$ to find the sizes of the intersections $C_i \cap C_k, k < i$.

9. set *score(k)* equal to $|C_i \cap C_k|, k < i$. The number of elements in the intersection is given by summing the vector representing $C_i \cap C_k$.

10. end internal k dependent for loop.

11. begin if test for empty intersection, so that no edge is added when all intersections $C_i \cap C_k, k < i$ are empty.

12. if the maximum of score is greater than zero, then find j, the position of the maximum. Note that routine argmax.m returns the index of the first maximal element in the case of nonuniqueness.

13. add the edge (C_i, C_j) to the junction tree by setting the i, jth and j, ith entries of the adjacency matrix *jt* equal to one.

14. end the internal if test.

15. end outer i dependent for loop.

```
function [jtree]=ripcliques_to_jtree_zo(cliques)
% input:  1. cliques, a p x p matrix array respresentation of a
%            perfect sequence of cliques of g.
%            (eg, from chordal_to_ripcliques_zo.m)

% output: jtree, the adjacency matrix of a junction tree
%         (not necessarily unique) with respect to cliques

% WARNING: this algorithm only works for cliques outputted
% with respect to a maximum cardinality search ordering of
% the cliques, as then the cliques are already ordered with
```

```
% respect to the rank of the highest vertex in the clique.
% For example, if mcs ordering of vertices is [6 5 3 1 4 2], then a
% clique comprised of {6,1} will precede a clique comprised of {4,2}
% since 6 and 1 precede 4 and 2 in the ordering [6 5 3 1 4 2].

   p=size(cliques, 1);
   clique_sizes=zeros(1,p);
   clique_sizes=sum(cliques, 1);
       % this is a 1 x p vector of column totals.
       % summing the column cliques(:,i) gives the number of vertices
       % in clique C_i.
   num_cliques=size(find(clique_sizes), 2);
   score=zeros(1,p);
   jtree=zeros(p,p);

   for i=2:num_cliques;
      for k=1:i-1;
        score(k)=sum(intersect_zo(cliques(:,i), cliques(:,k)) );
      end;

      if max(score)~=0 ;
        % only add the edge if clique i IS connected to one of its
        % predecessors. if score is all zeros, then clique has no intersection
        % with any of its predecessors. Since the cliques are in RIP, it must
        % follow that we are no longer in the same connected component of
        % the graph.
        % if clique i has no intersection with any of the preceding cliques,
        % then the graph is disconnected, so the adjacency matrix will have
        % a zero row/column for this i, and we have a forest, not a j_tree.

        j=argmax(score);
        jtree(i,j)=1; jtree(j,i)=1;
       end;
   end;
```

8.1.4 finding the separators, given a perfect sequence of cliques and the associated junction tree

Let $\mathcal{T}$ be a junction tree with the cliques of g as its vertices. Recall that by definition of a junction tree, the intersection $C_i \cap C_j$ is a subset of every clique on the necessarily unique path in $\mathcal{T}$ connecting C_i and C_j. Therefore, the intersections between adjacent cliques in $\mathcal{T}$ will have the largest number of elements.

Recall that a perfect sequence of sets $C_1, \ldots, C_k$ has the running intersection property. That is, for every $i > 1$, the intersection $S_i = C_i \cap H_{i-1} \subset C_j$ for some $j < i$. Therefore, the principle used to create the junction tree $\mathcal{T}$ from a perfect sequence of cliques ensures

that the intersections $C_i \cap C_j$ between adjacent cliques in jt will be equal to the separator sets S_i by the following argument.

Fact 1. the definition of the junction tree that the intersection between any pair must be a subset of the cliques on the necessarily unique path between them implies that the adjacent intersections are the biggest possible intersections which include either one of the adjacent pair. Fact 2. The junction tree jt was formed by adding an edge between C_i and any preceding clique which had maximal intersection with C_i. Fact 3. From the running intersection property, we know that $S_i \subset C_j, j < i$. Fact 4. But S_i is the intersection between C_i and the union of *all* cliques which precede C_i in the perfect sequence. Facts 1, 2, 3 and 4 together imply that $S_i = C_i \cap C_j$, and so S_i is the biggest intersection. Since we know that adjacent cliques have a (not necessarily unique) maximal intersection, then the separators S_i can be calculated as the intersection between adjacent cliques in the junction tree.

Note that if two cliques C_{j1} and C_{j2} each have same maximal intersection with C_i, then either of these could have been made adjacent to C_i in jt. This situation corresponds to the situation in which two of the sequence separators are the same, as illustrated by the decomposable $g5$ of Figure 3.14, in which all three cliques are separated by the same sequence separator, $S_k = \{a, b\}, k = 1, 2$.

Therefore, the sets which separate C_i and C_j can be found by taking the pairwise intersections of adjacent cliques in the junction tree. Furthermore, the intersections $S_i = C_i \cap C_j$ of adjacent cliques will separate $C_i \backslash C_i \cap C_j$ and $C_j \backslash C_i \cap C_j$ in g.

An alternative to this routine that does not require the input of a junction tree is given in Section 8.1.10.

The following 13 item list description is enumerated with respect to the 13 lines of MATLAB code (excluding comment and blank lines) that follows it. This code finds the separators and their sizes from a perfect sequence of cliques and the associated junction tree, based on the above described principle of adjacent cliques having maximal intersections which are the sequence separators and the graph separators. The cell array version is described before giving an alternative matrix array version.

1. find $t =$ number of vertices in *jtree*, the junction tree. This is the same as the number of cliques of g.

2. initialise a $t \times t$ array *sepsize* to zero. *sepsize(i,j)* will equal $|C_i \cap C_j|$ if C_i and C_j are adjacent, and will remain zero if they are not adjacent.

3. initialise *seps*, a $t \times t$ cell array for storing the associated separators $C_i \cap C_j$ for

adjacent cliques.

4. begin for loop $i = 1, \ldots t$ to find the clique adjacent to C_i.

5. begin internal for loop $j = i + 1, \ldots t$ to consider all cliques $C_j, j > i$ for adjacency test. Note that this will find all adjacent pairs, as the adjacency matrix $jtree$ is symmetric so only the upper (or lower) triangular half needs to be searched.

6. begin if test so that intersections are only taken between cliques adjacent in $\mathcal{T}$, as given by the nonzero entries of the adjacency matrix $jtree$.

7. if the pair of cliques C_i and C_j are adjacent, take the intersection $C_i \cap C_j$ and store as the ijth entry in *seps*.

8. if the pair of cliques C_i and C_j are adjacent, find $|C_i \cap C_j|$ and store as the ijth entry in *sepsize*.

9. make *seps* symmetric.

10. make *sepsize* symmetric.

11. end the internal if test.

12. end inner j dependent for loop.

13. end outer i dependent for loop.

```
function [sepsize, seps]=separators_cell(cliques, jtree)
% NOTE: if you just want a 1 x num_seps array of the sequence separators, use
% seps_residuals_histories.m

% inputs: 1. cliques, a 1 x t cell array of the t nonempty cliques of g in
%            RIP ordering (from chordal_to_ripcliques_cell.m)
%         2. jtree, the associated t x t adjacency matrix of the junction tree

% output: 1. sepsize, a matrix array of the size of the separator sets, in which
%            sepsize(i,j) = [number of elements in intersection between
%            cliques cliques{i} and cliques{j} if they are adjacent, and zero else.
%         2. seps, a (num_cliques)x(num_cliques) cell array
%            in which seps{i,j}=cliques{i} intersect cliques{j}.

   t=size(cliques,2);
   sepsize=zeros(size(jtree)); % =num_cliques x num_cliques
   seps=cell(size(jtree));

   for i=1:t;
```

```
      for j=i+1:t;
        if jtree(i,j)==1;
           seps{i,j}=intersect(cliques{i}, cliques{j});
           sepsize(i,j)=length(intersect(cliques{i}, cliques{j}));

% this version does the full matrix/cell array
% which is symmetric so actually unnecessary.
% But could be dangerous not to compute in case the wrong ordering
% j, i, for j > i is used by one of the calling programs)

           seps{j,i}=seps{i,j};
           sepsize{j,i}=sepsize(i,j);
        end;
      end;
   end;
```

The matrix array code is now explained.

1. find p, the maximum possible number of vertices in *jtree*, the junction tree. This is the same as the size of the array *cliques*.

2. initialise a $p \times p$ array *sepsize* to zero. *sepsize(i,j)* will equal the $|C_i \cap C_j|$ if C_i and C_j are adjacent, and will remain zero if they are not adjacent.

3. initialise to zeros a $1 \times p$ vector *clique_sizes* to record the number of variables in each clique. This can be computed by summing each column of *cliques*, since $cliques(i, j) = 1$ if and only if $v_i \in C_j$.

4. calculate the column totals and store in *clique_sizes*.

5. calculate *num_cliques*, the number of nonempty cliques by finding how many of the p entries in *clique_sizes* are nonzero.

6. begin for loop $i = 1, \ldots$ *num_cliques* to find the clique adjacent to C_i.

7. begin internal for loop $j = i + 1, \ldots$ *num_cliques* to consider all cliques $C_j, j > i$ for adjacency test. Note that by the symmetry of the adjacency matrix for the junction tree, only the upper (or lower) triangular half needs to be searched to find all adjacent pairs.

8. begin if test so that intersections are only taken between cliques adjacent in $\mathcal{T}$, as given by the nonzero entries of the adjacency matrix *jtree*.

9. if the pair of cliques C_i and C_j are adjacent, take the intersection $C_i \cap C_j$ by using the subroutine *intersect_zo*. This returns a vector of zeros and ones representing the set of variables in the intersection. Hence the sum of this vector gives the number of variables in the intersection. This sum is stored as the ijth entry in *sepsize*.

10. make *sepsize* symmetric.

11. end the internal if test.

12. end inner j dependent for loop.

13. end outer i dependent for loop.

```
function [sepsize]=sepsize_zo(cliques, jtree)
% NOTE: if you just want a 1 x num_seps array of all the
% sequence separators, use seps_residuals_histories.m

% inputs:  1. cliques, a p x p matrix array respresentation of a
%             perfect sequence of cliques of g.
%             (eg, from chordal_to_ripcliques_zo.m)
%          2. jtree, the associated p x p adjacency matrix
%             of the junction tree

% output: 1. sepsize, a matrix array of the size of the
%             separator sets, in which
%             sepsize(i,j) = |C_i intersection C_j| between
%             adjacent cliques C_i and C_j with respect to jtree.

   [p]=size(cliques,1);
   sepsize=zeros(p,p);
   clique_sizes=zeros(1,p);
   clique_sizes=sum(cliques, 1);
        % this is a 1 x p vector of column totals
   num_cliques=size(find(clique_sizes),2);

   for i=1:num_cliques;
      for j=i+1:num_cliques;
        if jtree(i,j)==1;
            sepsize(i,j)=sum(intersect_zo(cliques(:,i), cliques(:,j)));

% this version does the full matrix/cell array
% which is symmetric so actually unnecessary.
% But could be dangerous not to compute in case the wrong ordering
% j, i, for j > i is used by one of the calling programs)

            sepsize(j,i)=sum(seps_ij);
         end;
      end;
   end;
```

8.1.5 finding the path matrix of g, in which the i, jth entry is one if vertices v_i and v_j are connected

It is well known that for any $p \times p$ adjacency matrix g, the powers g^k are the k-step transition matrices. The longest possible path without repetition of vertices is $p-1$. Hence the finite sum sum $g + g^2 + ... + g^{p-1}$ is the transition matrix indicating all the vertices that are connected in $1 \leq k \leq p-1$ transition steps. Each ijth entry of the sum shows the total number of possible paths of length $1 \leq k \leq p-1$ between v_i and v_j, and any zero entry indicates that v_i and v_j are not connected.

The following 9 item list description is enumerated with respect to the 9 lines of MATLAB code (excluding comment and blank lines) that follows it. The code computes the transition matrix by computing the sum described above.

1. find p, the number of vertices in g.
2. initialise A as g, the single step transition matrix.
3. initialise *reach_graph*, the $p \times p$ array of the number of paths length $\leq p-1$ between vertex pairs.
4. begin for loop $i = 1, \ldots p-1$ for the sequential sum.
5. add the next i-step transition matrix to *reach_graph*.
6. calculate the next $(i+1)$-step transition matrix.
7. end for loop $i = 1, \ldots p-1$.
8. replace every entry greater than one with a one. The MATLAB command $C = (C >$ 0) returns a matrix with 1s in every place where C ¿ 0.

```
function reach_graph = reachability_graph(g)
% inputs: 1. g, the p x p symmetric adjacency matrix with
%               respect to an original ordering v_1, ..., v_p

% output: 1. reach_graph, a p x p symmetric matrix, in which
%            reach_graph(i,j) = 1 iff there is a path from i to j

% This computes g = g + g^2 + ... + g^{p-1}, which is the transition matrix
% showing all possible vertices that can be
% reached in 1, ..., p-1 steps. Note that the longest
% possible path is p-1, so only need to consider up to and including paths
% of length p-1
```

```
p = size(g,1);
A = g;
reach_graph = zeros(p);
for i=1:p-1
   reach_graph = reach_graph + A;
   A = A * g;
end
reach_graph = (reach_graph > 0);
     % C = (C > 0) gives a matrix with 1s in every place where C > 0.
```

8.1.6 finding the set of neighbours of a single vertex

Since g is the adjacency matrix of a graph on vertices $V = \{v_1, \ldots v_p\}$, the indicies of the neighbours of v_i are the indicies of the nonzero entries of row i, as returned by the MATLAB *find* function, and the $1 \times |nbrs(v_i)|$ array output will represent the set of neighbours in the cell array version.

The following single item list description is for the single line of MATLAB code (excluding comment and blank lines) that follows it.

1. use inbuilt MATLAB find function to return the indicies of all vertices j such that $g_{ij} = 1$. Since g is the adjacency matrix, this is the same as the indicies of all vertices adjacent to v_i. Note that MATLAB will return the indicies of the vertices in ascending order.

```
function [ns]=neighbours_vertex_cell(g, i)
% inputs: 1. g, the p x p symmetric adjacency matrix with
%               respect to an original ordering v_1, ..., v_p

          2. i, the index with respect to g, of the vertex
             v_i whose neighbours are required.

% output: ns, a 1 x |nbrs(v_i)| array of indicies of nbrs(v_i).
          Returns the empty vector if v_i has no neighbours

   ns=find(g(i,:));

% NOTE: MATLAB find will return the vertices in ascending order
```

The matrix array version must return a $p \times 1$ vector ns, such that $ns(j) = 1$ whenever v_j is adjacent to v_i. Therefore the neighbours are given by the ith column of g. The following 3 item list description is for the 3 lines of MATLAB code (excluding comment and blank lines) that follows it.

1. find p, the number of vertices.
2. initialize ns, the $p \times 1$ vector of zeros and ones to represent the set $nbrs(v_i)$.
3. $nbrs(v_i)$ is given by the ith column of the adjacency matrix g, since by definition of the adjacency matrix g, $g_{ij} = 1$ if and only if v_i and v_j are adjacent, and $g_{ij} = 0$ otherwise.

```
function [ns]=neighbours_vertex_zo(g, v_i)
% inputs: 1. g, the p x p symmetric adjacency matrix with
%            respect to an original ordering v_1, ..., v_p

          2. i, the index with respect to g, of the vertex
             v_i whose neighbours are required.

% output: ns, a p x 1 vector of zeros and ones, representing nbrs(v_i).
          Returns zero vector if no neighbours.

    p=size(g,1);
    ns=zeros(p,1);
    ns=g(:,i);

% NOTE: MATLAB find will return the vertices in ascending order
```

8.1.7 finding the set of parents of a single vertex

This routine assumes that the ordering by which the parents are neighbouring predecessors, is the ordering $1, \ldots p$ of the adjacency matrix matrix g.

The following 3 item list description is enumerated with respect to the 3 lines of MATLAB code (excluding comment and blank lines) that follows it. This code finds $pa(v_i)$ as the intersection between the vertices $v_1, \ldots, v_{i-1}$ and $nbrs(v_i)$. The cell array version is described before giving an alternative matrix array version.

1. initialise ps, the set of parents, as the empty set.
2. find ns, the set of indicies of the neighbours of v_i.
3. find ps, the set of indicies of $pa(v_i)$, by taking the intersection of all indicies preceding i, and the indicies of $nbrs(v_i)$. Note that MATLAB will return the indicies of the vertices in ascending order.

```
function [ps]=parents_vertex_cell(g, i)
% inputs: 1. g, the p x p symmetric adjacency matrix with
```

```
%               respect to an original ordering v_1, ..., v_p

          2. i, the index with respect to g, of the vertex
             v_i whose parents are required.

% output: ps, a 1 x |pa(v_i)| array of indicies of pa(v_i) with
          respect to the ordering 1, 2,..., p.
          Returns the empty vector if v_i has no parents.

%% NOTE: This only works if vertices are ordered as per g.
%%        i.e. if mcs order is 1, 7, 3, 4, 2, ...
%%        then it will return
%%        parents(adj_mat, 4)=intersect([1 7 3 4], [1:4])
%%                                    =[1 3]
%%        which is WRONG

   ps=[];
   ns=neighbours_node_cell(g, i);
   ps=intersect([1:i], ns);
```

The matrix array version must return a $p \times 1$ vector ps, such that $ps(j) = 1$ whenever v_j is adjacent to v_i and precedes v_i in the order $1, \ldots, p$, and $ps(j) = 0$ otherwise. Therefore the parents are represented by the vector ps defined by $ps(j) = 0, j \geq i$ and $ps(j) = 1$ for all indicies less than i which are also included in the vector of indicies of $nbrs(v_i)$. The following 5 item list description is for the 5 lines of MATLAB code (excluding comment and blank lines) that follows it.

1. find p, the number of vertices in g.
2. initialise to the zero vector ps, the set of indicies of the parents.
3. find the indicies of the neighbours of v_i.
4. set the indicies of the parents to be the indicies of the neighbours.
5. make zero all the indicies which follow i.

```
function [ps]=parents_vertex_zo(g, i)
% inputs: 1. g, the p x p symmetric adjacency matrix with
%            respect to an original ordering v_1, ..., v_p

          2. i, the index with respect to g, of the vertex
             v_i whose neighbours are required.

% output  1: ps, a p x 1 vector representing pa(v_i)
%
% NOTE: This only works if vertices are ordered as per g.
```

```
%       i.e. if mcs order is 1, 7, 3, 4, 2, 6, 5
        then it will return
%          [1 0 1 0 0 0 0]'=parents(adj_mat, 4) (=[1,3])
%       which is WRONG (should be [1 0 1 0 0 0 1]'={1 7 3})

   p=size(g, 1);
   ps=zeros(p,1);
   nbs=neighbours_vertex_zo(g, i);
   ps=nbs;
   ps(i:p,1)=0;
```

8.1.8 finding the first clique in a perfect sequence that contains a given vertex v_a

This code finds the index of the first clique in *cliques* that contains the vertex v_a. Note that v_a must be contained in at least one clique or the routine never breaks. The cell array version is described before the matrix array version.

The following 11 item list description of the cell array version is enumerated with respect to the 11 lines of MATLAB code (excluding comment and blank lines) that follows it.

1. initialise $i = 1$ as the index of the clique to be searched for the vertex v_a.
2. initialise the variable *found* to zero. *found* will be set to one when a clique containing v_a is found.
3. initialise *index_a*, the index of the clique containing v_a, to that of the first clique.
4. begin conditional *while* loop so that the routine continues to search for the clique containing v_a until *found* is set to one.
5. begin internal j dependent for loop, to test if v_j is an element of the ith clique.
6. begin internal if test to see if the jth element of the ith clique is equal to v_a, by testing to see if *cliquesi(j)=a*.
7. if *cliquesi(j)=a*, then set $index\_a = i$, found=1, and break out of the internal for loop so that v_i is not tested against any further elements of the ith clique.
8. begin internal if test to increment the index of the clique being searched if v_a is still unfound after comparison with all elements of the ith clique.

9. if v_a is still not found, then add one to the index of the clique to be searched, to look for v_a in the next clique in the sequence.

10. end internal if test to increment the index of the cliques.

11. end conditional *while* loop.

```
function [index_a]=find_clique_containing_cell(a, cliques)
% inputs: 1. a, the index of the vertex v_a with respect
%          to the ordering of cliques.
%         NOTE: if v_a is not in one of the cliques, this routine
%          will go into an infinite while loop.
%         2. cliques, a cell array of nonempty cliques

% output: index_a, the index of the first clique
%         (in the order of the cell array)
%         that contains v_a

   i=1;
   found=0;
   index_a=1;
   while found==0;
      for j=1:length(cliques{i});
         if cliques{i}(j)==a;
            index_a=i; found=1; break; end,
         end;
        if found == 0;
            i=i+1;
        end
   end;
```

The matrix array version is trivial. Since the $p \times p$ array *cliques* is such that $cliques_{ij} = 1$ if and only if $v_i \in C_j$, and $cliques_{ij} = 0$ otherwise, the indicies of all cliques containing v_a is given by the indicies of the nonzero entries in the ath row of *cliques*. The following 4 item list description of the matrix array version is enumerated with respect to the 4 lines of MATLAB code (excluding comment and blank lines) that follows it.

1. find p, the maximum possible number of cliques.

2. initialise the $p \times 1$ variable *all_indicies* to zero. *all_indicies* is the vector of indicies of all the cliques that contain v_a.

3. use the MATLAB inbuilt *find* function to find *all_indicies*.

4. use the inbuilt MATLAB *min* function to set *index_a* as the minimum index, and hence the first clique in the ordering of *cliques* that contains v_a.

```
function [index_a]=find_clique_containing_zo(a, cliques)
% inputs: 1. index_a, the index of the vertex v_a with respect
%            to the ordering of cliques.
%            NOTE: if v_a is not in one of the cliques, this routine
%            will go into an infinite while loop.
%         2. cliques, a p x p matrix array respresentation of a
%            perfect sequence of cliques of g.
%            (eg, from chordal_to_ripcliques_zo.m)

% output: index_a, the index of the first clique (with respect to the
          column order of the array cliques) that contains v_a

   p=size(cliques,1);
   all_indicies=zeros(p,1);
   all_indicies=find(cliques(a, :));
   index_a=min(all_indicies);
```

8.1.9 finding all cliques in a perfect sequence that contains a given vertex v_a

This code finds all the indicies of the cliques in *cliques* that contain the vertex v_a. The cell array version is given before the matrix array version.

1. initialise *index_a*, the vector such that *index_a*(i)=1 if and only if $v_a \in C_i$.
2. begin internal i dependent for loop, to test for existence in the ith clique.
3. begin internal j dependent for loop, to test if v_j is an element of the ith clique.
4. begin internal if test to see if the jth element of the ith clique is equal to v_a, by testing to see if *cliquesi(j)=a*.
5. if *cliquesi(j)=a*, then set $index\_a = i$.
6. end conditional if to find $index\_a = i$.
7. end j dependent for loop to test if $v_j \in C_i$.
8. end i dependent for loop to test all C_i.

```
function [all_indicies_a]=find_all_clique_containing_cell(a, cliques)
% input: 1. a, the index of v_a wrt adjacency matrix g
%        2. cliques, a cell representation of cliques of g.
% output:1. all_indicies_a, the indicies (wrt order of cliques) of all cliques
%        that contain v_a
```

```
index_a=zeros(1, length(cliques));
for i=1:length(cliques)
    for j=1:length(cliques{i});
       if cliques{i}(j)==a;
        index_a(i)=i;
       end;
    end
end

all_indicies_a=find(index_a);
```

The matrix array version is trivial. Since the $p \times p$ array *cliques* is such that $cliques_{ij} = 1$ if and only if $v_i \in C_j$, and $cliques_{ij} = 0$ otherwise, the indicies of all cliques containing v_a is given by the indicies of the nonzero entries in the ath row of *cliques*.

1. use the MATLAB inbuilt *find* function to find *all_indicies*.

```
function [all_indicies_a]=find_all_clique_containing_zo(a, cliques)
% input: 1. a, the index of v_a wrt adjacency matrix g
%        2. cliques, a p x p matrix representation of cliques of g.
% output:1. all_indicies_a, the indicies (wrt order of cliques) of all cliques
%        that contain v_a

   all_indicies_a=find(cliques(a, :));
```

8.1.10 finding the sets of separators, residuals and histories, given a perfect sequence of cliques

Subsection 2 introduced the sets of subsets of V denoted $\mathcal{S}, \mathcal{R}, \mathcal{H}$, respectively, of sequence separators, residuals and histories. The code explained below finds these sets of subsets given $\mathcal{C}$, a perfect sequence, using the definitions given in Subsection 2. Note that unlike the code in Section 8.1.4, this code does not require the input of a junction tree, so can be used to find the separators without calculating a junction tree. The code is first explained with respect to each set of sets having a cell array representation, then followed by the equivalent code based on a matrix array representation.

The following 11 item list description is enumerated with respect to the 11 lines of code (excluding comment and blank lines) that follows it.

1. find *num_cliques*, the number of cliques.
2. define *num_seps*, as $|\mathcal{S}|$ assuming that the first separator is empty.
3. find *num_non_empty_seps*. This is $|\mathcal{S}| - 1$.

4. initialise *seps*, the cell array of separators, to a cell array of empty sets.
5. initialise *resids*, the cell array of residuals, to a cell array of empty sets.
6. initialise *hists*, the cell array of histories, to a cell array of empty sets.
7. calculate the first history, *hists{1,1}* equal to the first clique.
8. begin *for* loop, dependent on *index*$= 2, \ldots, |\mathcal{C}|$. The initialisation of the cell arrays, together with the loop beginning at *index=2*, ensures that both the first separator and residual are each empty.
9. calculate the *index*th hisory as the union of the first *index* cliques.
10. calculate the *index*th separator as the intersection of the *index*th clique and the *(index-1)*th history.
11. calculate the *index*th residual as the set difference between the *(index-1)*th history and the *index*th clique.

```
function [seps, resids, hists]=seps_resids_hists_cell(cliques)
% input:    1. cliques, a 1x|num_cliques| cell array of RIP ordered cliques

% output:   1. seps, a 1x (num_cliques) cell array of the
%              separators wrt the ordering of the cell array cliques.
%              Note that by definition, the indexes j of S_j begin at 2,
%              so below defines the first separator seps{1,1}=[].
%           2. resids, a 1x (num_cliques) cell array of the
%              histories wrt the ordering of the cell array cliques.
%              Note that by definition, the indexes j of R_j begin at 2,
%              so below defines the first residual resids{1,1}=[].
%           3. hists, a 1x (num_cliques) cell array of the
%              separators wrt the ordering of the cell array cliques.

   num_cliques=size(cliques,2);
   num_seps=num_cliques;
   num_non_empty_seps=num_seps-1;
   seps=cell(1, num_seps);
   resids=cell(1, num_seps);
        % NOTE i always set the first empty, as everyone
        % indexes j as 2,..., num_cliques.

   hists=cell(1, num_seps);
   hists{1,1}=cliques{1};

   for index=2:num_cliques;
      hists{1,index}=union(cliques{index}, hists{index-1});
```

```
      seps{1,index}=intersect(cliques{index}, hists{index-1});
      resids{1,index}=setdiff(cliques{index}, hists{index-1});
   end;

% Sj are intersection of total HISTORY and the
% new clique, not C_j and C_j-1.
% All of output ordered min:max by matlab.
```

The following 11 item list description of the matrix array version is enumerated with respect to the 11 lines of code (excluding comment and blank lines) that follows it.

1. find p, the maximum possible number of cliques.

2. calculate *num_cliques*, the number of nonempty cliques. Since the output array *cliques* from *chordal_to_ripcliques_zo.m* has a zero column if and only if it represents an empty clique, then the sum of each column is the number of elements in the associated clique. Hence the largest index of the columns which do not sum to zero is the number of nonempty cliques, and can be calculated by $max(find(sum(cliques, 1) \neq 0))$.

3. initialise to zero *seps* the matrix representation array of $\mathcal{S}$.

4. initialise to zero *resids* the matrix representation array of $\mathcal{R}$.

5. initialise to zero *hists* the matrix representation array of $\mathcal{H}$.

6. calculate the first history, *hists(:,1)=cliques(:,1).*

7.

8. begin for loop, dependent on *index*$= 2, \ldots, |\mathcal{C}|$. The initialisation of the matrix arrays, together with the loop beginning at *index=2*, ensures that both the first separator and residual are each empty.

9. calculate the *index*th hisory as the union of the first *index* cliques.

10. calculate the *index*th separator as the intersection of the *index*th clique and the *(index-1)*th history.

11. calculate the *index*th residual as the set difference between the *(index-1)*th history and the *index*th clique.

```
function [seps, resids, hists]=seps_resids_hists_zo(cliques)
% input:    1. cliques, a p x p matrix array respresentation of a
%              perfect sequence of cliques of g.

% output:   1. seps, a p x p matrix array of the
%              separators wrt the ordering of the matrix array cliques.
%              Note that by definition, the indexes j of S_j begin at 2,
%              so below defines the first separator as the empty set.
%           2. resids, a p x p matrix array of the
%              histories wrt the ordering of the matrix array cliques.
%              Note that by definition, the indexes j of R_j begin at 2,
%              so below defines the first residual as the empty set.
%           3. hists, a a p x p matrix array of the
%              separators wrt the ordering of the matrix array cliques.

   p=size(cliques,1);
   num_cliques=max(find( sum(cliques,1)~=0));
        % sum(cliques,1)) gives sum of each col=number of elts in clique
        % find...~=0 returns indicies of nonzero cols, so take max for num

   seps=zeros(p, p);
   resids=zeros(p, p);
      % NOTE i always set the first as empty, so index=2,..., num_cliques.
   hists=zeros(p, p);
   hists(:,1)=cliques(:,1);

   for index=2:num_cliques;
       hists(:,index)=union_zo(cliques(:,index), hists(:,index-1));
       seps(:,index)=intersect_zo(cliques(:,index), hists(:,index-1));
       resids(:,index)=setdiff_zo(cliques(:,index), hists(:,index-1));
   end;

% Sj are intersection of total HISTORY and the
% new clique, not Cj and Cj-1
```

8.1.11 checking legality of edge removals

This code is based on the principle that an edge cannot be legally deleted if it is in more than one clique, and can be otherwise. This characterisation of legal deletions is according to Lemma 3.3.2. By Theorem 4.4.1, the single clique C_q which contains the edge that can be legally deleted is required to define S_{q2} in Lemmas 4.4.3 and 4.4.5, and hence to calculate the ratio of graph marginal likelihoods for the MH transition probability. Therefore C_q is outputted by this routine. The code is first explained with respect to each set of sets having a cell array representation, then followed by the equivalent code based on a matrix array representation.

The following 14 item list description is enumerated with respect to the 14 lines of code

(excluding comment and blank lines) that follows it.

1. initialise to 'yes' the indicator of a legal removal, *delete_ok*.
2. initialise to zero the count of the number of cliques that contain the edge (v_i, v_j).
3. find t, the number of cliques.
4. begin for loop, dependent on $k = 1, cdots, t$, to find the cliques which contain (v_i, v_j).
5. let C_k be the representation of C_k, kth clique.
6. begin if test for the edge $(v_i, v_j) \in C_k$. The cell array representation ensures that if $(v_i, v_j) \in C_k$, then the intersection of the vector C_k with the vector *[i,j]* will equal *[i,j]*, so be of size 2.
7. if $(v_i, v_j) \in C_k$, then set C=C_k. The calculation of the likelihood depends on the clique which contains the edge which can be legally deleted, so it must be outputted by this routine.
8. if $(v_i, v_j) \in C_k$, add one to the count of the number of cliques containing (v_i, v_j).
9. begin if test for more than one clique containing (v_i, v_j).
10. if the there are strictly greater than one cliques containing the edge, then it is an illegal removal, so set the indicator *delete_ok* to zero, and a dummy return *C=[9 9 9]*.
11. if the edge is in 2 cliques, then it is illegal. There is no point testing for inclusion in other cliques, so exit the for loop.
12. end the test for $(v_i, v_j) \in C_k$.
13. end the test for more than one clique containing the edge.
14. end the external for loop.

```
function [delete_ok, C]=check_edge_delete_cell(i,j, cliques)
% inputs:  1. i, j, the indicies of the edge vertices with respect
%             to the original ordering in the adjacency matrix g
%          2. cliques, a 1 x t cell array of a perfect sequence of
%             (nonempty) cliques,
%             such as from chordal_to_ripcliques_cell.m

% outputs: 1. delete_ok=1/0
```

```
%             (yes/no, can delete and remain chordal)
%    2. C=[clique which contained the legal edge],
%             as represented by a 1 x |C| array of vertex indicies
%             with respect to original g
%             The likelihood depends on C, so it is outputted by
%             this routine.

   delete_ok=1;
   count_cliques=0;
       % initialise to 0 the count of cliques containing the edge
   t=size(cliques,2);

   for k=1:t;
      C_k=cliques{k};
      if length(intersect([i,j], C_k))==2;
         % since cliques are by definition complete,
         % if a clique contains the ith and jth vertices,
         % then it contains the edge (v_i,v_j).
         C=C_k;
            % if legal delete, this is clique edge is in.
        count_cliques=count_cliques +1;
        if count_cliques > 1;
            delete_ok =0; C=[9 9 9];
            break;
        end;
      end;
   end;

% Theory: can ONLY delete edges that are in a single clique
% (else in separator). Draw picture and can see why.
```

The following 14 item list description of the matrix array version is enumerated with respect to the 14 lines of code (excluding comment and blank lines) that follows it.

1. find p, the number of vertices and the maximum possible number of cliques.

2. initialise to 'yes' the indicator of a legal removal, *delete_ok*.

3. initialise to zero C, the vector representation of the clique which contains the edge (v_i, v_j).

4. initialise to zero *count_cliques*, the count of the number of cliques that contain the edge (v_i, v_j).

5. calculate the vector *clique_sizes*, in which the ith entry is the size of the ith clique. This is given by the column sums of the zero one matrix array representation, *cliques*.

6. calculate *num_cliques*, the number of cliques. This is the number of nonzero elements of *clique_sizes*, so equal to the number of indicies returned by the MATLAB *find* function along the second dimension of *clique_sizes*.

7. initialise to zero *edge_col_vec*, a $p \times 1$ column vector representation of the edge (v_i, v_j) with respect to the ordering of the original adjacency matrix g.

8. set the jth element of *edge_col_vec* to 1 to represent that v_i is a member of the edge.

9. set the jth element of *edge_col_vec* to 1 to represent that v_j is a member of the edge.

10. begin for loop, dependent on $k = 1, cdots$,*num_cliques*, to find the cliques which contain (v_i, v_j).

11. let *clique_k* be the representation of C_k, kth clique.

12. begin if test for the edge $(v_i, v_j) \in C_k$. If $(v_i, v_j) \in C_k$, then there will be a one in the ith and jth positions of the vector representation of the intersection. Hence the sum of the intersection will equal 2.

13. if $(v_i, v_j) \in C_k$, then set *C=clique_k*. The calculation of the likelihood depends on the clique which contains the edge which can be legally deleted, so it must be outputted by this routine.

14. if $(v_i, v_j) \in C_k$, add one to the count of the number of cliques containing (v_i, v_j).

15. begin if test for more than one clique containing (v_i, v_j).

16. if the there are strictly greater than one cliques containing the edge, then it is an illegal removal, so set the indicator *delete_ok* to zero, and a dummy return *C=[9 9 9]*.

17. if the edge is in 2 cliques, then it is illegal. There is no point testing for inclusion in other cliques, so exit the for loop.

18. end the test for $(v_i, v_j) \in C_k$.

19. end the test for more than one clique containing the edge.

20. end the external for loop.

```
function [delete_ok, C]=check_edge_delete_zo(i,j, cliques)
% inputs:  1. i, j, the indicies of the edge vertices with respect
%             to the original ordering in the adjacency matrix g
%          2. cliques, a p x p matrix array respresentation of a
%            perfect sequence of cliques of g.
%            (eg, from chordal_to_ripcliques_zo.m)

% outputs: 1. delete_ok=1/0
%             (yes/no, can delete and remain chordal)
%    2. C=[clique which contained the legal edge],
%             as represented by a p x 1 array of zeros and ones
%             with respect to original g.
%             The likelihood depends on C, so it is outputted by
%             this routine.

   p=size(cliques,1);
   delete_ok=1;
   C=zeros(p,1);
   count_cliques=0;
       % initialise to 0 the count of cliques containing the edge
   clique_sizes=zeros(1,p);
   clique_sizes=sum(cliques, 1);
      % this is a 1 x p vector of column totals
   num_cliques=size(find(clique_sizes),2);

%%%%%% create p x 1 vector representation of edge
   edge_col_vec=zeros(p,1);
   edge_col_vec(i,1)=1;
   edge_col_vec(j,1)=1;

   for k=1:num_cliques;
      clique_k=cliques(:,k);
      if sum(intersect_zo(edge_col_vec, clique_k))==2;
         % since cliques are by definition complete,
        % if a clique contains the ith and jth vertices,
         % then it contains the edge (v_i,v_j).
         C=clique_k;
             % if legal delete, this is clique edge is in.
         count_cliques=count_cliques +1;
         if counter > 1;
            delete_ok =0; C=[9 9 9];
            break;
         end;
      end;
   end;

% Theory: can ONLY delete edges that are in a single clique
% (else in separator). Draw picture and can see why.
```

8.1.12 checking legality of edge additions using Theorem 2, Giudici & Green (1999)

This code is based on the Giudici & Green (1999) characterisation that an edge can be added legally between vertices v_i and v_j if and only if there exists a pair of cliques $C_i, C_j \in \mathcal{C}$ such that $v_i \in C_i, v_j \in C_j$ and $C_i \cap C_j$ is equal to a separator on the path between C_i and C_j in the associated junction tree. Lemma 3.3.4 gives an equivalent characterisation of legal additions that is not based on junction trees, making most of the code obselete. However, the code was written before Lemma 3.3.4 and the results are based on the below code. Note that the routine is only called by the main program if the vertices are in the same connected component, as the edge addition is trivially legal when the vertices are in different connected components of g. Using the notation of Section 4.4, if g is the decomposable graph got from g' by adding the single edge e, then by Theorem 4.4.1, the clique C_q required to define S_{q2} in Lemmas 4.4.3 and 4.4.5, and hence to calculate the ratio of graph marginal likelihoods for the MH transition probability, is equal to the union $e \cup \{C_u \cap C_v\}$ for the cliques C_u, C_v of Lemma 3.3.4. Therefore C_q is outputted by this routine.

The code is first explained with respect to the cell array representation of sets, then followed by the equivalent code based on a matrix array representation. The calculation of the likelihood depends on C which consists of the union of the vertices in e and the vertices in the intersection of the cliques in the characterisation. Therefore C must be outputted by this routine.

The code is first explained with respect to each set of sets having a cell array representation, followed by the explanation of the equivalent code based on a matrix array representation. Because the program is so long, the code will be explained in sections, each corresponding to a specific purpose. The itemised list description will be enumerated with respect to the lines of code (excluding comment and blank lines) of the section rather than the entire program. For clarity and ease of matching description with code, at the end of each section the corresponding last line of that section's code will be inserted.

1. initialise to 'no' the indicator of a legal addition, *edge_ok*.

2. initialise to 'no' the indicator *quit* that the routine can be aborted. *quit* will be set equal to one for the first pair of cliques that satisfy the conditions of the characterisation.

3. initialise to zero the index of the fork of the tree, *fork*.

4. initialise to 'no' the indicator *CASE_same_branch* that the vertices are on the same branch of the tree.

5. initialise to 'no' the indicator *CASE_sats_on_b2_branch* that the characterisation is satisfied on the branch of the tree that contains the *index_b2*th clique in the perfect sequence represented by the cell array *cliques*.

6. initialise to 'no' the indicator *locate_a2* that a clique containing v_a, the ath vertex (with respect to the original adjacency matrix g), has been located.

7. find t, the number of cliques.

```
t=size(cliques,2);  % END 1st SECTION for code description
```

The second section finds the indicies of the first pair of cliques in the perfect sequence that contain v_a and v_b, the ath and bth vertices of the graph, respectively. It then renames the clique indicies and the vertices so that: the *index_b2*th clique C_{index_b2} contains v_{index_b2}, the *index_b2*th vertex of the graph; the *index_a2*th clique C_{index_a2} contains v_{index_a2}, the *index_a2*th vertex of the graph; and *index_b2* is strictly greater than *index_a2*. Since C_{index_b2} is the first clique in the perfect sequence that contains v_{index_b2}, and since *index_b2* is strictly greater than *index_a2*, then C_{index_b2} must be closer to any clique containing v_{index_a2} than any other clique containing v_{index_b2}. If v_{index_a2} is on the same branch of the tree as v_{index_b2}, then the closest clique containing v_{index_a2} will be the first clique containing v_{index_a2} on a path from C_{index_b2} upwards towards the root of the junction tree. Note that in this case, the closest clique containing v_{index_a2} will not necessarily be C_{index_a2}. If two or more cliques contain v_a2, then the closest clique to C_{index_b2} will be the *last* clique in the perfect sequence that contains v_{a2}, and C_{index_a2} was the first. Based on this fact, the code tests if $v_{a2} \in C_k$ for all the cliques C_k that are on the path from C_{index_b2} to the root, and exits the loop as soon as the first clique containing v_{a2} is found.

1. find *index_b*, the index with respect to *cliques* of the first clique in the perfect sequence that contains v_b.

2. find *index_a*, the index with respect to *cliques* of the first clique in the perfect sequence that contains v_a.

3. let *index_b2* be the greater of *index_a* and *index_b*.

4. if *index_b2* is equal to *index_b*, then set $b2 = b, a2 = a$.

5. otherwise, set $b2 = a, a2 = b$.

6. end if test for renaming indicies.

7. clear the original *index_a*.

8. let *index* equal *index_b2*. This will be the actual index of the cliques C_k with respect to the perfect sequence represented by *cliques*.

9. find *next_index*, the index of the single parent of *C_index* with respect to the junction tree *jtree*. This parent must be unique because *jtree* is a tree.

10. begin for loop, dependent on $dummy = 1, cdots$, *index_b2*-1, to look for v_{a2} in every clique on the path to the top of *jtree*, allowing for the maximum possible length path.

11. begin if test for reaching the top of the tree and not finding a clique containing v_{a2}. At the top of the tree, the parent of the current clique is empty.

12. if the top of the tree is reached, and the indicator for finding a clique containing v_{a2} is still zero, then exit the loop and set *CASE_same_branch*$= 0$ to indicate that v_{a2} is not on the same branch as v_{b2}.

13. end if test for reaching the top of the tree and not finding a clique containing v_{a2}.

14. begin if test for $v_{a2} \in C_{next\_index}$ and the parent index being well defined. Note that the test *isempty(next_index)*$==0$ ensures that the parent is well defined; i.e. that the very first clique C_{index_b2} is not the top of the tree. If C_{index_b2} is the top of the tree, then, the first *next_index* calculated is the empty set, and the variable *cliques*$\{$*next_index*$\}$ is not defined. So an error message will result if the test is only for $v_{a2} \in C_{next\_index}$.

15. if $v_{a2} \in C_{next\_index}$ and C_{next_index} is defined, then set *index_a2*=*next_index*, set the indicators for locating v_{a2} and v_{a2} being on the same branch as v_{b2} both one (for 'yes'), and exit the for loop as the closest clique containing v_{a2} has been found.

16. if $v_{a2} \notin C_{next\_index}$, set *index*=*next_index*.

17. find the parent of the new C_{index}. This clique is the next clique on the path. Call this parent the new C_{next_index}.

18. end for loop looking for v_{a2} in every clique on the path to the top of *jtree*.

```
end;  % END 2nd SECTION for code description
```

If v_{a2} was not located on the path from C_{index_b2} to the top of the tree, then it must be the case that there is a fork on the path between C_{index_b2} and the closest clique to it that contains v_{a2}. In this case, unlike the previous case, the closest clique is the one closest to the top of the tree, and will be the first in the perfect sequence. Therefore *index_a2 will be* the output of the routine *find_clique_containing(a2, cliques).m*, even though this was not the case if the cliques were on the same branch of *jtree*. The final section of code tests the characterisation that the intersection of the closest cliques C_{index_b2} and C_{index_a2} is a separator between them. In practice, only the size of the intersection needs to be tested because (by the definition of a junction tree) $C_{index\_a2} \cap C_{index\_b2} \subset C_k$ for every clique C_k on the path.

1. begin if test for indicator *CASE_same_branch*$= 0$, indicating that C_{index_a2} was not on the same branch as C_{index_b2}.

2. if C_{index_a2} was not on the same branch as C_{index_b2}, then there is a fork on the shortest path between them, so C_{index_a2} is the first clique in the perfect sequence containing v_{a2}.

3. end if test for same branch.

4. find the intersection of the closest cliques, as represented by the indicies in $C_{index\_a2} \cap C_{index\_b2}$ with respect to the adjacency matrix g.

5. calculate $s = |C_{index\_a2} \cap C_{index\_b2}|$.

6. if s is zero, then the intersection is empty and cannot be a separator in any connected component of g. So return 'no', not legal, by setting the indicator *edge_ok*$= 0$. In this case, set C equal to the empty set, and the indicator *quit*$= 1$ so that the routine can be exited prematurely.

7. begin if test for the cliques being adjacent in *jtree*. By the construction of *jtree*, adjacent cliques have intersection equal to a separator.

8. if the cliques are adjacent, return 'yes', legal, by setting the indicator *edge_ok*$= 1$. In this case, set C as the union of the edge vertices and the intersection, and the indicator *quit* to 1 so that the routine can be exited prematurely.

9. end if test for continuing the routine and the cliques being on the same branch.

```
end;  % END 3rd SECTION for code description
```

If the indicator *quit* is still zero, then all the separators on the path between C_{index_a2} to C_{index_b2} must be tested for equality with $C_{index\_a2} \cap C_{index\_b2}$. As already noted, only the size of the intersection needs to be tested. If at any stage there exists a separator of size s, then the characterisation of legal addition is satisfied so the routine can be exited prematurely.

The next two sections check that the closest pair of cliques satisfies the characterisation that their intersection is a separator on the path in *jtree* between them. Two cases need to be considered. The first case is if the cliques were on the same branch. In this case, the path is from C_{index_a2} to C_{index_b2} via parents. The comparison is therefore between s and the size of all the pairwise intersections of adjacent cliques in *jtree*, on the path from C_{index_b2} to C_{index_a2}.

1. begin if test for *quit* still zero and C_{index_a2} on the same branch as C_{index_b2}.
2. in this case, define *bottom=index_b2*.
3. in this case, define *next_parent_b2* as the first parent of C_{index_b2} in *jtree*.
4. begin *while* loop to terminate once the next parent is C_{index_a2}; i.e. continue while *next_parent_b2*$\geq$ *index_a2*.
5. if *next_parent_b2*$\geq$ *index_a2*, then begin if test for comparing s with the size of the separator between $C_{next\_parent\_b2}$ and C_{bottom}.
6. if s the size of the separator between $C_{next\_parent\_b2}$ and C_{bottom} is equal to s, then the characterisation is satisfied. Return *edge_ok*= 1, and $C = \{v_a, v_b\} \cup (C_{index\_a2} \cap C_{index\_b2})$.
7. set *quit* to exit the routine prematurely, and exit the while loop.
8. end if test for the intersection being a separator.
9. if the size of the next separator on the path is not equal to s, then redefine *bottom* as the current *next_parent_b2*,
10. in this case, define the new *next_parent_b2* to be the parent of the new C_{bottom} in *jtree*.
11. end *while* loop for testing all the separators on the path.

```
end; % END 4th SECTION for code description
```

The fifth section of code is for the second case in which C_{index_a2} and C_{index_b2} are not on the same branch, and none of the premature exit cases have been satisfied. In this case, the cliques are separated by a fork. Therefore the comparison is between s and the size of all the pairwise intersections of adjacent cliques in *jtree* on the 2 paths from C_{index_a2} to the fork, and C_{index_b2} to the fork. It is important that the comparison test terminates when the fork clique is reached, and not before or after. It is therefore necessary to find the fork clique. The fork clique need not be at the top of the tree, so the test for empty parents cannot be used as a condition for being at the fork clique. The below code finds the fork clique by finding the intersection of the set of indicies of the cliques on the path from C_{index_a2} to the top of the tree with the set of indicies of the cliques on the path from C_{index_b2} to the top of the tree. The maximum such index will be the index of the fork on the shortest path between C_{index_a2} and C_{index_b2} in *jtree*. In a junction tree of a perfect sequence constructed using the code in this chapter, every clique on the path from C_j to the top of the tree must have index less than j in the perfect sequence.

1. begin $elseif$ for the case where C_{index_a2} and C_{index_b2} are not on the same branch.

2. if quit is still zero and the cliques are not on the same branch, define *bottom_a2=index_a2*, and *next_parent_a2* as the parent of C_{index_a2} in *jtree*.

3. similarly, define *bottom_b2=index_b2*, and *next_parent_b2* as the parent of C_{index_b2} in *jtree*.

4. initialise to zero a $1\times$ *index_a2* vector *ancestors_a2* to record the indicies of C_{index_a2} and all the cliques on the path from C_{index_a2} to the top of the tree, allowing for the maximum possible number of cliques on this path. Similarly for *_b2*.

5. set the first *next_ancestor_a2* to be *index_a2* to record C_{index_a2} as the first clique on the path, and similarly for *_b2*.

6. begin for loop dependent on *count*$= 1, \ldots,$*index_a2* to find the indicies of the cliques on the path from C_{index_a2} to the top of the tree.

7. define the *count*th entry of *ancestors_a2* as *next_ancestor_a2* to represent that $C_{next\_ancestor\_a2}$ is the *count*th clique on the path.

8. find the new $C_{next\_ancestor\_a2}$ as the parent of the current $C_{next\_ancestor\_a2}$.

9. if the new $C_{next\_ancestor\_a2}$ is empty, then the current $C_{next\_ancestor\_a2}$ is at the top of the tree, so exit prematurely.

10. end for loop to find the indicies of the ancestors of C_{index_a2}.

11. use the inbuilt MATLAB function *nonzeros* to return only the nonzero entries of *ancestors_a2*.

12. begin for loop dependent on *count*$= 1, \ldots,$*index_b2* to find the indicies of the cliques on the path from C_{index_b2} to the top of the tree.

13. define the *count*th entry of *ancestors_b2* as *next_ancestor_b2* to represent that $C_{next\_ancestor\_b2}$ is the *count*th clique on the path.

14. find the new $C_{next\_ancestor\_b2}$ as the parent of the current $C_{next\_ancestor\_b2}$.

15. if the new $C_{next\_ancestor\_b2}$ is empty, then the current $C_{next\_ancestor\_b2}$ is at the top of the tree, so exit prematurely.

16. end for loop to find the indicies of the ancestors of C_{index_b2}.

17. use the inbuilt MATLAB function *nonzeros* to return only the nonzero entries of *ancestors_b2*.

18. find *fork_set*, the indicies of the set of all cliques which are ancestors of both C_{index_a2} and C_{index_b2}.

19. calculate *fork*, the fork on the shortest path, as the maximum of *fork_set*.

`% END 5th SECTION for code description` The final section first checks if the characterisation is satisfied on the path from C_{index_b2} to the fork clique. It calculates the correct number of separators to test as the number of clique indicies contained in *ancestors_b2* that exceed the index of the fork clique, thus ensuring that the search does not go above the fork clique. If the characterisation is still not satisfied, it does the same for the path from C_{index_b2} to the fork clique. This completes the testing of all cases in which the characterisation could possibly be satisfied.

1. calculate *num_seps_branch_b2*, the number of separators on the path from C_{index_b2} to the fork clique (which is the number of pairs to be tested).

2. begin for loop dependent on *count*$= 1, \ldots,$ *num_seps_branch_b2*, to test s for equality with the size of each separator on the path.

3. find the indicies of the next pair of adjacent cliques on the path.
4. begin if test for the characterisation being satisfied for this pair of cliques.
5. if s is the size of the separator between the next pair of adjacent cliques, then the characterisation is satisfied. Return *edge_ok*= 1, and $C = \{v_a, v_b\} \cup (C_{index\_a2} \cap C_{index\_b2})$.
6. set *quit* to exit the routine prematurely, and the indicator that the characterisation is satisfied on the branch of the fork that includes C_{index_b2}. Exit the for loop prematurely.
7. end the if test for the characterisation.
8. end the for loop for testing the intersection of each pair of cliques on the path.
9. begin if test that the routine is not yet to be exited, and that the characterisation was not satisfied on the branch of the fork that includes C_{index_b2}.
10. in this case, calculate the *num_seps_branch_a2*, the number of separators on the path from C_{index_a2} to the fork clique (which is the number of pairs to be tested).
11. begin for loop dependent on *count*= $1, \ldots,$ *num_seps_branch_a2*, to test s for equality with the size of each separator on the path.
12. find the indicies of the next pair of adjacent cliques on the path.
13. begin if test for the characterisation being satisfied for this pair of cliques.
14. if s is the size of the separator between the next pair of adjacent cliques, then the characterisation is satisfied. Return *edge_ok*= 1, and $C = \{v_a, v_b\} \cup (C_{index\_a2} \cap C_{index\_b2})$.
15. set *quit* to exit the routine prematurely, and the indicator that the characterisation is satisfied on the branch of the fork that includes C_{index_b2}. Exit the for loop prematurely.
16. end the if test for the characterisation.
17. end the for loop for testing the intersection of each pair of cliques on the path.
18. end the begin if test that the routine is not yet to be exited, and that the characterisation was not satisfied on the branch of the fork that includes C_{index_b2}.

19. end the external $elseif$ test for C_{index_a2} and C_{index_b2} not being on the same branch of *jtree*.

20. if the premature exit indicator is still zero, then since all cases have been exhausted, the edge addition is illegal. Return *edge_ok*= 0, and C equal to the emptyset.

```
% END 6th and final SECTION for code description
```

```
function [edge_ok, C]=check_edge_add_same_component_cell(a,b, jtree, sepsize, cliques)
% inputs:  1. a, b, the indicies of the edge vertices with respect
%             to the original ordering in the adjacency matrix g
%          2. jtree, the adjacency matrix of a junction tree
%             with respect to cliques.
%          3. sepsize, a matrix array of the size of the separator sets, in which
%             sepsize(i,j) = [number of elements in intersection between
%             cliques cliques{i} and cliques{j} if they are adjacent, and zero else.
%          4. cliques, a 1 x t cell array of a perfect sequence of
%             (nonempty) cliques,
%             such as from chordal_to_ripcliques_cell.m

% outputs: 1. edge_ok=1/0
%             (yes/no, can add and remain chordal)
%         2. C, the new clique which contains the vertices of the added edge,
%             as represented by a 1 x |C| array of vertex indicies
%             with respect to original g
%             The likelihood depends on C, so it is outputted by
%             this routine.

%%% NOTE: this routine is ONLY called by the main program if
%%% the verticies are in the same connected component.
   edge_ok=0;
   quit = 0;
   fork=0;
   CASE_same_branch=0;
   CASE_sats_on_b2_branch =0;
   locate_a2=0;
   t=size(cliques,2);   % END 1st SECTION for code description

   index_b=find_clique_containing_cell(b, cliques);
   index_a=find_clique_containing_cell(a, cliques);

%%% Re_name verticies so b is in clique{index_b2}, a is in clique{index_a2}
   index_b2=max([index_a,index_b]);
      % index_b2 >1, as it is greater of the two, and
      % since a,b is NOT an edge, can't have index_a2=index_b2
   if index_b2==index_b; b2=b; a2=a;
     else b2=a; a2=b;
   end

   clear index_a;
```

```
index=index_b2;
next_index=parents_vertex_cell(jtree, index);
     % tree, so parent is a single vertex, not a vector

for dummy=1: index_b2-1;
    % perform the loop b2-1 times if no break

   if  isempty(next_index)==1 & locate_a2 ==0;
      CASE_same_branch=0; break,
    end
     % if next_index is empty AND locate_a2 ==0,
     % then at the top of the tree and KNOW
     % that index_a2 and index_b2 are on separate branches.
     % BUT can't assume the fork is top of tree.
     % Could be case of 3-2-4-5 with 1-2 the top of tree.

   if is_in(a2, cliques{next_index}) ==1 & isempty(next_index) ==0;
       index_a2=next_index; locate_a2= 1; CASE_same_branch=1; break,
   end
       % if you find a2 before you get to the top of the tree,
       % then KNOW that index_a2 and index_b2 are on same
       % branch of tree. So exit the loop, and set
       % locate_a2 case indicator.

    index=next_index;
    next_index=parents_vertex_cell(jtree, index);

   % if a2 is in any clique on the same side of the root
   % vertex in jtree, then that clique and the first clique
   % in the RIP ordering containing b2 will be the
   % end points of the shortest path between 2 containing
   % cliques for the verticies of the edge considered.
   % Performs the loop at least b2-1 times if no break,
   % which is the longest possible path to the top of a
   % connected component of the possibly disconnected tree.

end;  % END 2nd SECTION for code description

if CASE_same_branch==0,
   index_a2=find_clique_containing_cell(a2, cliques);
end
   % IF CASE_same_branch==0, (so locate_a2==0) then
   % must be fork between them. So index_a2 for shortest path
   % is first clique in RIP ordering containing a2.

intersection_of_cliques=intersect(cliques{index_a2}, cliques{index_b2});
s=length(intersection_of_cliques);
   % find intersection, so can test to see if
   % it is a separator. In practice, only need to
   % check that the size of this intersection is
```

```
      % equal to the size of a separator on this path.

   if s==0; edge_ok=0; C=[]; quit=1; end
      % the empty set is not a separator in a connected
      % tree, so condition cannot be satisfied. exit at this stage.

   if jtree(index_a2, index_b2)==1;
      edge_ok=1; C=union( [a,b], intersection_of_cliques); quit=1;
   end   % END 3rd SECTION for code description
               % if the cliques are adjacent in the tree, their
               % intersection is by definition a separator so exit.

%%% if quit still zero, next test all the separators
%%% between clique{index_a2} and clique{index_b2}
%%% at any stage, if there exists a separator of length s
%%% then know edge legal, so set edge_ok=1, quit=1, and exit loop.

   if ( (quit ==0) & (CASE_same_branch==1) );
        % first consider where a2 is on same branch as b2. only need to
        % test path from b2 to a2
      bottom=index_b2;
      next_parent_b2=parents_vertex_cell(jtree, index_b2);

      while ((next_parent_b2 >= index_a2) )
         if sepsize(next_parent_b2, bottom) == s;
             edge_ok=1; C=union( [a,b], intersection_of_cliques);
             quit=1; break,
          end,
                % only need to test equality of size,
                % since intersection contained in every intermediate
                % clique by RIP and junction tree property
        bottom=next_parent_b2;
        next_parent_b2 = parents_vertex_cell(jtree, bottom);
     end; % END 4th SECTION for code description

   elseif ( (quit==0) & (CASE_same_branch==0) );
      % if on different branches, have to test from index_b2 and
      % index_a2 to fork between them. CANNOT go to edge beyond fork.
      % Safest strategy is to find the fork.

        bottom_a2=index_a2; next_parent_a2=parents_vertex_cell(jtree, index_a2);
        bottom_b2=index_b2; next_parent_b2=parents_vertex_cell(jtree, index_b2);
        ancestors_a2=zeros(1, index_a2); ancestors_b2=zeros(1, index_b2);
        next_ancestor_a2=index_a2; next_ancestor_b2=index_b2;

        for count=1:index_a2
           ancestors_a2(count)=next_ancestor_a2;
           next_ancestor_a2 =parents_vertex_cell(jtree, next_ancestor_a2);
           if isempty(next_ancestor_a2), break, end
        end
```

```
      ancestors_a2=nonzeros( ancestors_a2)';
            % convert to only the ancestors using existing matlab function.

      for count=1:index_b2
         ancestors_b2(count)=next_ancestor_b2;
         next_ancestor_b2 =parents_vertex_cell(jtree, next_ancestor_b2);
         if isempty(next_ancestor_b2), break, end
      end

      ancestors_b2=nonzeros(ancestors_b2)';
             % convert to only the ancestors using existing matlab function.
      fork_set=intersect(ancestors_a2, ancestors_b2);
      fork=max(fork_set);
             % the clique where the two paths up the tree meet must be the
             % highest index in the intersection to the root.

      % END 5th SECTION for code description

      num_seps_branch_b2=length(find(ancestors_b2>fork));

    for count=1: num_seps_branch_b2
         index_row=ancestors_b2(count+1); index_col=ancestors_b2(count);
            % sepsize is lower diagonal zero, and the ancestors are stored
            % in descending order [index_b2,..., 1]
        if sepsize( index_row, index_col) == s;
             edge_ok=1; C=union( [a,b], intersection_of_cliques);
             quit=1; CASE_sats_on_b2_branch=1; break,
         end,
     end

    if (quit==0 & CASE_sats_on_b2_branch==0)
         % don't want to perform the above loop
         % if found separator=intersection.
       num_seps_branch_a2=length(find(ancestors_a2>fork));

       for count =1:num_seps_branch_a2
           index_row=ancestors_a2(count+1); index_col=ancestors_a2(count);
              % sepsize is lower diagonal zero, and the ancestors are stored
              % in descending order [index_a2,..., 1]
           if sepsize( index_row, index_col) == s;
               edge_ok=1; C=union( [a,b], intersection_of_cliques);
               quit=1; break,
           end,
       end
    end
end

if quit==0; edge_ok=0; C=[]; end
% END 6th and final SECTION for code description
```

Matrix array version

The itemised list description of the matrix array version below is sectioned in the same way as the cell array description. It is enumerated with respect to the lines of code (excluding comment and blank lines) of the corresponding section rather than the entire program.

1. initialise to 'no' the indicator of a legal addition, *edge_ok*.
2. initialise to 'no' the indicator *quit* that the routine can be aborted. *quit* will be set equal to one for the first pair of cliques that satisfy the conditions of the characterisation.
3. initialise to zero the index of the fork of the tree, *fork*.
4. initialise to 'no' the indicator *CASE_same_branch* that the vertices are on the same branch of the tree.
5. initialise to 'no' the indicator *CASE_sats_on_b2_branch* that the characterisation is satisfied on the branch of the tree that contains the *index_b2*th clique in the perfect sequence represented by the cell array *cliques*.
6. initialise to 'no' the indicator *locate_a2* that a clique containing v_a, the ath vertex (with respect to the original adjacency matrix g), has been located.
7. find p, the maximum possible number of cliques.

```
p=size(cliques,1);  % END 1st SECTION for code description
```

The second section has the same purpose as the second section of the cell array code given above. The only difference is in the representation of sets.

1. initialise to zero *ab_edge_vec*, the $p \times 1$ representation of the edge $e = (a, b)$.
2. set *ab_edge_vec(a)=1* and *ab_edge_vec(a)=1* to represent that $v_a, v_b \in e$.
3. find *index_b*, the index with respect to *cliques* of the first clique in the perfect sequence that contains v_b.
4. find *index_a*, the index with respect to *cliques* of the first clique in the perfect sequence that contains v_a.
5. let *index_b2* be the greater of *index_a* and *index_b*.

6. if *index_b2* is equal to *index_b*, then set $b2 = b, a2 = a$.

7. otherwise, set $b2 = a, a2 = b$.

8. end if test to rename the indicies.

9. clear the original *index_a*.

10. let *index* equal *index_b2*. This will be equal to each of the indicies of the cliques C_k with respect to the perfect sequence represented by *cliques*.

11. find *next_index*, the index of the single parent of *C_index* with respect to the junction tree *jtree*. This parent must be unique because *jtree* is a tree.

12. begin for loop, dependent on $dummy = 1, cdots$, $index\_b2-1$, to look for v_{a2} in every clique on the path to the top of *jtree*, allowing for the maximum possible length path.

13. begin if test for reaching the top of the tree and not finding a clique containing v_{a2}. At the top of the tree, the parent of the current clique is empty, so the sum of all entries in its zero one vector will be zero.

14. if the top of the tree is reached, and the indicator for finding a clique containing v_{a2} is still zero, then exit the loop and set *CASE_same_branch*$= 0$ to indicate that v_{a2} is not on the same branch as v_{b2}.

15. end if test for reaching the top of the tree and not finding a clique containing v_{a2}.

16. find *clique_next_index*, the $1 \times p$ zero one representation of the *next_index*th clique in the perfect sequence.

17. begin if test for $v_{a2} \in C_{next\_index}$ and the parent C_{next_index} being well defined. Note that the test *sum(next_index)*$== 0$ means that there is a nonzero entry in *next_index*, so that the parent is well defined; i.e. that the very first clique C_{index_b2} is not the top of the tree. If C_{index_b2} is the top of the tree, then, the first *parents_vertex_zo(jtree, index)* will be a $p \times 1$ array of zeros. Hence *next_index=find(parents_vertex_zo(jtree, index))*$= 0$, so *clique_next_index=cliques(:,0)* and results in an error message. So an error message can result if the test is only for $v_{a2} \in C_{next\_index}$.

18. if $v_{a2} \in C_{next\_index}$ and C_{next_index} is defined, then set *index_a2=next_index*, set the indicators for locating v_{a2} and v_{a2} being on the same branch as v_{b2} both one (for 'yes'), and exit the for loop as the closest clique containing v_{a2} has been found.

19. if $v_{a2} \notin C_{next\_index}$, set *index*=*next_index*.

20. find the parent of the new C_{index}. This clique is the next clique on the path. Call this parent the new C_{next_index}.

21. end *for* loop looking for v_{a2} in every clique on the path to the top of *jtree*.

```
end;  % END 2nd SECTION for code description
```

The third section has the same purpose and works on the same principles as the equivalent cell array version already described.

1. begin *if* test for indicator *CASE_same_branch*= 0, indicating that C_{index_a2} was not on the same branch as C_{index_b2}.

2. if C_{index_a2} was not on the same branch as C_{index_b2}, then there is a fork on the shortest path between them, so C_{index_a2} is the first clique in the perfect sequence containing v_{a2}.

3. end *if* test for same branch.

4. find *clique_a2_int_clique_b2*, the zero/one $p \times 1$ representation with respect to the original ordering of the adjacency matrix g of the intersection of the closest cliques, $C_{index\_a2} \cap C_{index\_b2}$.

5. calculate $s = |C_{index\_a2} \cap C_{index\_b2}|$. The zero/one representation implies that s is given by the sum of *clique_a2_int_clique_b2*.

6. if s is zero, then the intersection is empty and cannot be a separator in any connected component of g. So return 'no', not legal, by setting the indicator *edge_ok*= 0. In this case, set C equal to the empty set, and the indicator *quit*= 1 so that the routine can be exited prematurely.

7. begin *if* test for the cliques being adjacent in *jtree*. By the construction of *jtree*, adjacent cliques have intersection equal to a separator.

8. if the cliques are adjacent, return 'yes', legal, by setting the indicator *edge_ok*= 1. In this case, set C as the union of the edge vertices and the intersection, and the indicator *quit* to 1 so that the routine can be exited prematurely.

9. end *if* test for continuing the routine and the cliques being on the same branch.

```
end;  % END 3rd SECTION for code description
```

The next two sections have the same purpose and work on the same principles as the equivalent cell array versions already described. The first section deals with the first case; i.e. that the cliques are on the same branch.

1. begin if test for *quit* still zero and C_{index_a2} on the same branch as C_{index_b2}.
2. in this case, define *bottom=index_b2*.
3. in this case, define *next_parent_b2* as the first parent of C_{index_b2} in *jtree*. The index of this parent clique will be given by the position of the only nonzero entry in *parents_vertex_zo(jtree, index_b2)* (as returned by the MATLAB find function).
4. begin *while* loop to terminate once the next parent is C_{index_a2}; i.e. continue while *next_parent_b2*$\geq$ *index_a2* and the next parent is nonempty. If the next parent clique is nonempty, the sum of the entries in its zero/one representation is nonzero.
5. if *next_parent_b2*$\geq$ *index_a2*, and the next parent is nonempty, then begin if test for comparing s with the size of the separator between $C_{next\_parent\_b2}$ and C_{bottom}.
6. if s the size of the separator between $C_{next\_parent\_b2}$ and C_{bottom} is equal to s, then the characterisation is satisfied. Return *edge_ok*= 1, and $C = \{v_a, v_b\} \cup (C_{index\_a2} \cap C_{index\_b2})$.
7. set *quit* to exit the routine prematurely, and exit the while loop.
8. end if test for the intersection being a separator.
9. if the size of the next separator on the path is not equal to s, then redefine *bottom* as the current *next_parent_b2*,
10. in this case, define the new *next_parent_b2* to be the parent of the new C_{bottom} in *jtree*.
11. end *while* loop for testing all the separators on the path.

```
end; % END 4th SECTION for code description
```

The fifth section of code is for the second case in which C_{index_a2} and C_{index_b2} are not on the same branch, and none of the premature exit cases have been satisfied. It works on the same principles as the equivalent cell array version already described.

1. begin $elseif$ for the case where C_{index_a2} and C_{index_b2} are not on the same branch.

2. if quit is still zero and the cliques are not on the same branch, define *next_parent_a2* as the parent of C_{index_a2} in *jtree*.

3. similarly, define *next_parent_b2* as the parent of C_{index_b2} in *jtree*.

4. initialise to zero a $1\times$ *index_a2* vector *ancestors_a2* to record the indicies of C_{index_a2} and all the cliques on the path from C_{index_a2} to the top of the tree, allowing for the maximum possible number of cliques on this path. Similarly for *_b2*.

5. initialise the corresponding $p \times 1$ zero/one representations.

6. set the first *next_ancestor_a2* to be *index_a2* to record C_{index_a2} as the first clique on the path, and similarly for *_b2*.

7. begin for loop dependent on *count*$= 1, \ldots,$*index_a2* to find the indicies of the cliques on the path from C_{index_a2} to the tree top.

8. define the *count*th entry of *ancestors_a2* as *next_ancestor_a2* to represent that $C_{next\_ancestor\_a2}$ is the *count*th clique on the path.

9. set the *next_ancestor_a2*th entry of the zero/one representation to one, so that it represents that $C_{next\_ancestor\_a2}$ is the *count*th clique on the path.

10. find the new $C_{next\_ancestor\_a2}$ as the parent of the current $C_{next\_ancestor\_a2}$.

11. if the new $C_{next\_ancestor\_a2}$ is empty (so the sum of its zero/one representation is zero), then the current $C_{next\_ancestor\_a2}$ is at the tree top, so exit prematurely.

12. end for loop to find the indicies from C_{index_a2} to the tree top.

13. begin for loop dependent on *count*$= 1, \ldots,$*index_b2* to find the indicies of the cliques on the path from C_{index_b2} to the tree top.

14. define the *count*th entry of *ancestors_b2* as *next_ancestor_b2* to represent that $C_{next\_ancestor\_b2}$ is the *count*th clique on the path.

15. set the *next_ancestor_b2*th entry of the zero/one representation to one, so that it represents that $C_{next\_ancestor\_b2}$ is the *count*th clique on the path.

16. find the new $C_{next\_ancestor\_b2}$ as the parent of the current $C_{next\_ancestor\_b2}$.

17. if the new $C_{next\_ancestor\_b2}$ is empty (so the sum of its zero/one representation is zero), then the current $C_{next\_ancestor\_b2}$ is at the top of the tree, so exit prematurely.

18. end for loop to find the indicies from C_{index_b2} to the tree top.

19. find *fork_set*, the zero/one vector which is equal to one at every position corresponding to the index of a clique vertex in both paths to the tree top.

20. the index of the fork clique is given by the greatest index amongst the nonzero entries in *fork_set*.

`% END 5th SECTION for code description` The final section has the same purpose and works on the same principles as the cell array version already described.

1. calculate *num_seps_branch_b2*, the number of separators on the path from C_{index_b2} to the fork clique (which is the number of pairs to be tested).

2. begin for loop dependent on *count*$= 1, \ldots,$ *num_seps_branch_b2*, to test s for equality with the size of each separator on the path.

3. find the indicies of the next pair of adjacent cliques on the path.

4. begin if test for the characterisation being satisfied for this pair of cliques.

5. if s is the size of the separator between the next pair of adjacent cliques, then the characterisation is satisfied. Return *edge_ok*$= 1$, and $C = \{v_a, v_b\} \cup (C_{index\_a2} \cap C_{index\_b2})$.

6. set *quit* to exit the routine prematurely, and the indicator that the characterisation is satisfied on the branch of the fork that includes C_{index_b2}. Exit the for loop prematurely.

7. end the if test for the characterisation.

8. end the for loop for testing the intersection of each pair of cliques on the *_b2* path.

9. begin if test that the routine is not yet to be exited, and that the characterisation was not satisfied on the branch of the fork that includes C_{index_b2}.

10. in this case, calculate the *num_seps_branch_a2*, the number of separators on the path from C_{index_a2} to the fork clique (which is the number of pairs to be tested).

11. begin for loop dependent on *count*$= 1, \ldots,$ *num_seps_branch_a2*, to test s for equality with the size of each separator on the path.

12. find the indicies of the next pair of adjacent cliques on the path.

13. begin if test for the characterisation being satisfied for this pair of cliques.

14. if s is the size of the separator between the next pair of adjacent cliques, then the characterisation is satisfied. Return *edge_ok*= 1, and $C = \{v_a, v_b\} \cup (C_{index\_a2} \cap C_{index\_b2})$.

15. set *quit* to exit the routine prematurely, and the indicator that the characterisation is satisfied on the branch of the fork that includes C_{index_b2}. Exit the for loop prematurely.

16. end the if test for the characterisation.

17. end the for loop for testing the intersection of each pair of cliques on the path.

18. end the begin if test that the routine is not yet to be exited, and that the characterisation was not satisfied on the branch of the fork that includes C_{index_b2}.

19. end the external $elseif$ test for C_{index_a2} and C_{index_b2} not being on the same branch of *jtree*.

20. if the premature exit indicator is still zero, then since all cases have been exhausted, the edge addition is illegal. Return *edge_ok*= 0, and C equal to the emptyset.

```
% END 6th and final SECTION for code description
```

```
function [edge_ok, C]=check_edge_add_same_component_zo(a,b, jtree, sepsize, cliques)
% inputs:  1. a, b, the indicies of the edge vertices with respect
%             to the original ordering in the adjacency matrix g
%          2. jtree, the adjacency matrix of a junction tree
%             (with respect to cliques) from ripcliques_to_jtree_zo.
%          3. sepsize, a matrix array of the size of the
%             separator sets, in which
%             sepsize(i,j) = |C_i intersection C_j| between
%             adjacent cliques C_i and C_j with respect to jtree.
%          4. cliques, a p x p matrix array respresentation of a
%             perfect sequence of cliques of g.
%             (eg, from chordal_to_ripcliques_zo.m)

% outputs: 1. edge_ok=0(no)/1(yes)
%          2. C, the p x 1 zero/one vector representation of the new clique
%             which contains the vertices of the added edge,
%             The likelihood depends on C, so it is outputted by
%             this routine.

%%% NOTE: this routine is ONLY called by the main program if
%%% the verticies are in the same connected component.
```

```
   edge_ok=0;
   quit = 0;
   fork=0;
   CASE_same_branch=0;
   CASE_sats_on_b2_branch =0;
   locate_a2=0;
   p=size(cliques,1);  % END 1st SECTION for code description

   ab_edge_vec=zeros(p,1);
   ab_edge_vec(a)=1; ab_edge_vec(b)=1;
   index_b=find_clique_containing_zo(b, cliques);
   index_a=find_clique_containing_zo(a, cliques);

%%% Re_name verticies so b is in cliques(index_b2, :), a is in cliques(index_a2, :);
%%% Note: jtree indicies correspond to verticies of cliques in RIP,
%%% i.e. order of vertex=clique is same as column index of cliques.

   index_b2=max([index_a,index_b]);
   if index_b2==index_b; b2=b; a2=a;
   else b2=a; a2=b;
   end
      % index_b2 >1, since max[ ]>1 (Can't have both a and b in first clique, as
      % there is no edge between them. Also, |index_a -index_b|>=1.
      % hence if..else ensures correct ordering of index_a2, index_b2
      % i.e. know clique(index_a2, :) precedes index_b2

   clear index_a;
   index=index_b2;
   next_index=find(parents_vertex_zo(jtree, index));
      % tree, so parent is unique
   for dummy=1: index_b2-1;    %this performs the loop at least b2-1 times if no break

      if  sum(next_index)==0 & locate_a2 ==0;
          % if parents empty (=[0 0 ... 0]') and not found a2, must be at tree top
          CASE_same_branch=0; break,
      end
         % if next_index is empty AND locate_a2 ==0, then at tree top
         % and KNOW index_a2 and index_b2 are on different branches. BUT can't
         % assume the fork is top of tree. could be case of 3-2-4-5 with 1-2
         % the top of tree.

      clique_next_index=cliques(:, next_index);

      if clique_next_index(a2,1) ==1 & sum(next_index) ~=0;
        % if a2 is in clique_next_index, and not at tree top
        index_a2=next_index; locate_a2=1; CASE_same_branch=1; break,
      end

         % if you find a2 before you get to the top of tree, then
```

```
        % KNOW index_a2 and index_b2 are on same branch of tree. So
        % break and set
        % locate_a2 case indicator.

        index=next_index;
        next_index=find(parents_vertex_zo(jtree, index));

     % if a2 is in any clique on the same side of the root
     % vertex in jtree, then that clique and the first clique
     % in the RIP ordering containing b2 will be the
     % end points of the shortest path between 2 containing
     % cliques for the verticies of the edge considered.
     % Performs the loop at least b2-1 times if no break,
     % which is the longest possible path to the top of a
     % connected component of the possibly disconnected tree.

  end;  % END 2nd SECTION for code description

  if CASE_same_branch==0,
     index_a2=find_clique_containing_zo(a2, cliques);
  end
     % IF CASE_same_branch==0, (so locate_a2==0) then
     % must be fork between them. So index_a2 for shortest path
     % is first clique in RIP ordering containing a2.

  clique_a2_int_clique_b2=intersect_zo(cliques(:,index_a2), cliques(:,index_b2));
  s=sum(clique_a2_int_clique_b2);
     % find intersection, so can test to see if
     % it is a separator. In practice, only need to
     % check that the size of this intersection is
     % equal to the size of a separator on this path.

  if s==0;
     edge_ok=0; C=zeros(p,1); quit=1;
  end
     % the empty set is not a separator in a connected
     % tree, so condition cannot be satisfied. exit at this stage.

  if jtree(index_a2, index_b2)==1;
     edge_ok=1; C=union_zo(ab_edge_vec, clique_a2_int_clique_b2); quit=1;
  end   % END 3rd SECTION for code description
           % if the cliques are adjacent in the tree, their
           % intersection is by definition a separator so exit.

%%% if quit still zero, next test all the separators
%%% between clique_a2 and clique_b2
%%% at any stage, if there exists a separator of length s
%%% then know edge add legal, so set edge_ok=1, quit=1, and exit loop.

  if ( (quit ==0) & (CASE_same_branch==1) );
```

```
       % first consider where a2 is on same branch as b2. only need to
       % test path from b2 to a2
      bottom=index_b2;
      next_parent_b2=find(parents_vertex_zo(jtree, index_b2));

      while (sum(next_parent_b2)>0 )  & (next_parent_b2 >= index_a2 )
          if sepsize(next_parent_b2, bottom) == s;
             edge_ok=1; C=union_zo(ab_edge_vec, clique_a2_int_clique_b2);
             quit=1;  break,
          end,
                % only need to test equality of size,
                % since intersection contained in every intermediate
                % clique by RIP and junction tree property
         bottom=next_parent_b2;
         next_parent_b2=find(parents_vertex_zo(jtree, bottom));
     end; % END 4th SECTION for code description

   elseif ( (quit==0) & (CASE_same_branch==0) );
      % if on different branches, have to test from index_b2 and
      % index_a2 to fork between them. CANNOT go to edge beyond fork.
      % Safest strategy is to find the fork.

%%% NOTE: in below, the parent clique vertex of is unique,
%%%       as only 1 parent in trees

      next_parent_a2=find(parents_vertex_zo(jtree, index_a2));
      next_parent_b2=find(parents_vertex_zo(jtree, index_b2));
      ancestors_a2=zeros(1, index_a2); ancestors_b2=zeros(1, index_b2);
         % ancestors_a2 elements are actual number indicies of clique ancestors
      ancestors_a2_col_vec=zeros(p,1); ancestors_b2_col_vec=zeros(p,1);
         % the col_vec equivalent
      next_ancestor_a2=index_a2; next_ancestor_b2=index_b2;

      for count=1:index_a2
         ancestors_a2(count)=next_ancestor_a2;
         ancestors_a2_col_vec(next_ancestor_a2)=1;
         next_ancestor_a2 =find(parents_vertex_zo(jtree, next_ancestor_a2));
         if sum(next_ancestor_a2)==0, break, end
      end

      for count=1:index_b2
         ancestors_b2(count)=next_ancestor_b2;
         ancestors_b2_col_vec(next_ancestor_b2,1)=1;
         next_ancestor_b2 =find(parents_vertex_zo(jtree, next_ancestor_b2));
         if sum(next_ancestor_b2)==0, break, end
      end

      fork_set=intersect_zo(ancestors_a2_col_vec, ancestors_b2_col_vec);
      fork=max(find((fork_set)));
              % need to find the place in the tree where the fork is, as defined
```

```
            % by the clique which is at this point of intersection
            % the clique where the two paths up the tree meet must be the
            % biggest index in the intersection to the root.
            % i.e. if they both share clique(:,3), then the path of
            % ancestors for both must include the ancestors of clique(:,3)
            % which if were clique2, clique1, then where the paths split is
            % at clique 3 = max(find(fork_set)))

        % END 5th SECTION for code description

      num_seps_branch_b2=length(find(ancestors_b2>fork));

%%%%  VERY IMPORTANT %%% only test separators between clique_fork and indicies

      for count=1: num_seps_branch_b2
         index_row=ancestors_b2(count); index_col=ancestors_b2(count+1);
            % sepsize is lower diagonal zero, and the ancestors are stored
            % in descending order [index_b2,..., 1]
        if sepsize( index_row, index_col) == s;
            edge_ok=1; C=union_zo( ab_edge_vec, clique_a2_int_clique_b2);
            quit=1; CASE_sats_on_b2_branch=1; break,
         end,
      end

      if (quit==0 & CASE_sats_on_b2_branch==0)
           % don't want to perform the above loop
           % if found separator=intersection.
         num_seps_branch_a2=length(find(ancestors_a2>fork));

         for count =1:num_seps_branch_a2
            index_row=ancestors_a2(count+1); index_col=ancestors_a2(count);
              % sepsize is lower diagonal zero, and the ancestors are stored
              % in descending order [index_a2,..., 1]
            if sepsize( index_row, index_col) == s;
               edge_ok=1; C=union_zo( ab_edge_vec, clique_a2_int_clique_b2);
               quit=1; break,
            end,
         end
      end
   end

   if quit==0; edge_ok=0; C=[]; end
   % END 6th and final SECTION for code description
```

8.1.13 checking legality of edge additions using Lemma 3.3.6

```
function [edge_ok, C]=check_edge_add_helen_cell(g, a,b, cliques)
edge_ok=0;
index_b=find_all_clique_containing(b, cliques);
```

```
index_a=find_all_clique_containing(a, cliques);
C=[];
int_cliques=[]; max_int=[];
size_int=0; max_size=0;
for i=1:length(index_a)
   for j=1:length(index_b)
        int_cliques=intersect(cliques{index_a(i)},cliques{index_b(j)});
        size_int=length(int_cliques);
        if size_int>max_size
         max_size=size_int;
         max_int=int_cliques;
        end
   end
end

  g_no_max_int=g;
  g_no_max_int(max_int, max_int)=0 ;

reach_graph_g_no_max_int=reachability_graph(g_no_max_int) ;
if reach_graph_g_no_max_int(a,b)==0
  edge_ok=1, C=union(max_int, [a,b]);
end

if edge_ok==0,
C=[];
end

% this will work in both connected and unconnected,
% since in unconnected
% the reach graph of g_no_max_int(a,b)= the reach graph g(a,b) = 0
```

8.1.14 calculating a g-constrained version of Σ.

The below code calculates the g-constrained version of any matrix B using the theory of Section 2.5. By Lemma 2.5.2 this can be calculated efficiently as follows. Invert each clique dependent subblock B_{C_j,C_j} and fill to full dimension with zeros. Sum these, then subtract the analogous sum of separator dependent subblocks, allowing for repetitions of the $S_j = C_j \cap H_{j-1}$. The full matrix B need not be inverted. The cell array version is described before the matrix array version. The itemised list descriptions are enumerated with respect to the lines of MATLAB code (excluding comment and blank lines) that immediately follow each.

1. find p, the number of variables and dimension of B.

2. calculate *seps*, the cell array of separators.

3. find *num_cliques*, the number of cliques.
4. initialise to zero *sum_big_K_cliques* and *sum_big_K_seps*, a $p \times p$ array of the cumulative sum of each term on the cliques and separators, respectively.
5. begin for loop $j = 2, \ldots$ *num_cliques* to add each partial sum.
6. clear all the previous j loop values.
7. let *cj* be the jth clique in the sequence.
8. find *B_cj*, the associated clique subblock of B.
9. calculate *K_cj*, the inverse of the jth clique subblock.
10. initialise to zero *big_Kj*, a temporary $p \times p$ array in which only those entries indexed by the jth clique are nonzero.
11. embed *K_cj* in *big_Kj*.
12. calculate the next partial sum by adding *big_Kj* to the previous partial sum.
13. end for loop to calculate the clique sum.

Lines 14 to 22 repeat the above process for the separators. Line 23 calculates *K_hat* as the difference in the partial sums *sum_big_K_cliques* − *sum_big_K_seps*, and Line 24 calculates $B\_hat = (K\_hat)^{-1}$.

```
function [B_hat, K_hat]=g_constrain_cell(B, cliques)
% g must be decomposable
% B, symmetric positive definite

% inputs: 1. B, the p x p symmetric matrix with
%            respect to an original ordering v_1, ..., v_p of g
%         2. cliques, a 1 x t cell array of a perfect sequence of
%            (nonempty) cliques of g,
%            such as from chordal_to_ripcliques_cell.m
% output: 1. B_hat, the g-constrained version of B
%         2. K_hat, the g-constrained version of inv(B_hat)


% THEORY: if (i,j) is edge or i=j, B_hat(i,j)=B(i,j)
% else B_hat satisfies [inv(B_hat)](i,j)=0
% such a B_hat for decomposable models has been
% shown to be unique
% (see either Grone et al 84, or
% Speed and Kiiveri, 1986, p. 142 Thereom 1.
```

```
% Calculate using pp. 144, 145 Lauritzen 96, based on
% Lemma 5.5 p.136 Lauritzen 96 for block sum expression.

  p=size(B,1);
  seps=seps_resids_hists(cliques);
  num_cliques=size(cliques, 2);
  sum_big_K_cliques=zeros(p); sum_big_K_seps=zeros(p);

  for j=1:num_cliques
    clear cj B_cj K_cj
    cj=cliques{j};
    B_cj=B(cj, cj);
    K_cj=inv(B_cj);
    big_Kj=zeros(p);
    big_Kj([cj], [cj])=K_cj;
    sum_big_K_cliques=sum_big_K_cliques+big_Kj;
  end

  for j=2:num_cliques
         %% NOTE i always set the first empty, as everyone numbers as 2, num_cliques.
    clear sj B_sj K_sj
    sj=seps{j};
    B_sj=B(sj, sj);
    K_sj=inv(B_sj);
    big_Kj=zeros(p);
    big_Kj([sj], [sj])=K_sj;
    sum_big_K_seps=sum_big_K_seps+big_Kj;
  end

  K_hat=sum_big_K_cliques-sum_big_K_seps;
  B_hat=inv(K_hat);
```

The matrix array code is now explained.

1. find p, the number of variables and dimension of B.
2. calculate *seps*, the matrix representation of the separators.
3. find *num_cliques*, the number of cliques. This is calculated as the maximum index of the columns of the matrix representation of the cliques with nonzero sum. Recall that the sum of a column will be zero if and only if it represents the empty set of variables.
4. initialise to zero *sum_big_K_cliques* and *sum_big_K_seps*, a $p \times p$ array of the cumulative sum of each term on the cliques and separators, respectively.
5. begin for loop $j = 2, \ldots$ *num_cliques* to add each partial sum.

6. clear all the previous j loop values.

7. let *cj* be the jth clique in the sequence. The actual indices representing the variables in the jth clique are needed for the embedding process. These are given by the MATLAB find function, as the indicies of the rows in the jth column of *cliques* that are not zero.

8. find *B_cj*, the associated clique subblock of B.

9. calculate *K_cj*, the inverse of the jth clique subblock.

10. initialise to zero *big_Kj*, a temporary $p \times p$ array in which only those entries indexed by the jth clique are nonzero.

11. embed *K_cj* in *big_Kj*.

12. calculate the next partial sum by adding *big_Kj* to the previous partial sum.

13. end for loop to calculate the clique sum.

Lines 14 to 22 repeat the above process for the separators. Line 23 calculates *K_hat* as the difference in the partial sums *sum_big_K_cliques* − *sum_big_K_seps*, and Line 24 calculates $B\_hat = (K\_hat)^{-1}$.

```
function [B_hat, K_hat]=g_constrain_zo(B, order, cliques)
% g must be decomposable
% B, symmetric positive definite

% inputs: 1. B, the p x p symmetric matrix with
%            respect to an original ordering v_1, ..., v_p of g
%         2. cliques, a p x p matrix array respresentation of a
%            perfect sequence of cliques of g.
%            (eg, from chordal_to_ripcliques_zo.m)
% output: 1. B_hat, the g-constrained version of B
%         2. K_hat, the g-constrained version of inv(B_hat)


% THEORY: if (i,j) is edge or i=j, B_hat(i,j)=B(i,j)
% else B_hat satisfies [inv(B_hat)](i,j)=0
% such a B_hat for decomposable models has been
% shown to be unique
% (see either Grone et al 84, or
% Speed and Kiiveri, 1986, p. 142 Thereom 1.
% Calculate using pp. 144, 145 Lauritzen 96, based on
% Lemma 5.5 p.136 Lauritzen 96 for block sum expression.

  p=size(B,1);
  seps=seps_resids_hists_zo(cliques);
```

```
num_cliques=max(find( sum(cliques,1)~=0));
  % sum(cliques,1)) gives sum of each col=numelts in clique
  % find..~=0 returns indicies of non-zero cols, so take max for num
sum_big_K_cliques=zeros(p); sum_big_K_seps=zeros(p);

for j=1:num_cliques
  clear cj B_cj K_cj
  cj=find(cliques(:,j))'; % note the transpose =cliques{j};
  B_cj=B(cj, cj);
  K_cj=inv(B_cj);
  big_Kj=zeros(p);
  big_Kj([cj], [cj])=K_cj;
  sum_big_K_cliques=sum_big_K_cliques+big_Kj;
end

for j=1:num_cliques %%% NOTE i always set the first empty, as everyone numbers as 2, num_cliques.
    clear sj B_sj K_sj
    sj=find(seps(:,j))'; %=seps{j};
    B_sj=B(sj, sj);
    K_sj=inv(B_sj);
    big_Kj=zeros(p);
    big_Kj([sj], [sj])=K_sj;
    sum_big_K_seps=sum_big_K_seps+big_Kj;
 end

 K_hat=sum_big_K_cliques-sum_big_K_seps;
 B_hat=inv(K_hat);
```

8.1.15 sampling $\Sigma \sim HIW(g, \delta, I)$.

The below code samples $\Sigma_{id} \sim HIW(g, \delta, I)$ based on Theorem 4.3.2 and the theory explained in Section 4.3. It first creates the perfect elimination scheme by reversing the order given by $C_1, R_2, \ldots, R_k$. Next, it permutes the ordering of the adjacency matrix g to the perfect elimination order. Theorem 4.3.2 can then be used to generate all the elements of the Cholesky of Σ^{-1} independently, and calculates the Σ_{id} with respect to the elimination order. The routine then permutes the rows and columns of the matrices generated back to the original ordering of the vertices of g.

The cell array version is described before the matrix array version. The itemised list descriptions are enumerated with respect to the lines of MATLAB code (excluding comment and blank lines) that immediately follow each.

1. initialise *index_finish* and *index_start*, the variables for indexing the sequence of vertices in *perfect_order*.

2. find *num_cliques*, the number of cliques.

3. find *residuals*, the set of residuals.

4. initialise *num_Rj*, a vector in which the ith entry is the number of elements in the jth residual. Note that the sequence of (nonempty) residuals begins at $j = 2$.

5. find p, the number of vertices in g.

6. initialise *perfect_order*, the $1 \times p$ permutation vector of indicies $1, \ldots p$ ($1, \ldots p$ is the original indexing in the adjacency matrix g) that gives the perfect numbering which when reversed is the permutation vector of a perfect elimination scheme.

7. order vertices in C_1 first in the perfect numbering.

8. set *index_finish* as the index of the last element entered in the sequence so far. This is $|C_1|$. Then *index_start* can be updated to the last index plus one in the loop that follows.

9. begin for loop, $j = 2, \ldots,$ *num_cliques* to reorder the vertices.

10. set Rj as the jth residual.

11. find $|Rj|$, and store as the jth entry in *num_Rj*.

12. update *index_start* to the previous *index_finish* plus one.

13. update *index_finish*. This is the index of the last vertex entered in the current $|Rj|$. It is given by the current *index_start* plus $|Rj|$ less one.

14. enter R_j as the *index_start*th to *index_finish*th vertices in the perfect sequence.

15. end for loop for reordering vertices.

16. initialise *rev_perf*, the $1 \times p$ permutation vector that gives the reverse perfect numbering required for a perfect elimination scheme.

17. initialise *index* to zero. This is a counter for putting the $p - index$th element of *perfect_order* at the ith position in *rev_perf*.

18. begin for loop, $i = 1, \ldots, p$ to create *rev_perf*.

19. the ith entry in the reverse seqence is the $p - index$th entry in *perfect_order*.

20. increment *index* by one.

21. end for loop to create *rev_perf*.

22. initialise *g_rev_perf*, the permuted version of g.

23. permute the rows and columns of g to give *g_rev_perf*. Note that to undo the permutation, the ith entry of perfect order is related to the jth entry in its reverse by the relation $i = p - j + 1$. That is, $i + j = p + 1$. For example, for the $p = 5$ sequence enumerations, we have $[1, 2, 3, 4, 5] + [5, 4, 3, 2, 1] = 6 = (p + 1)$.

24. initialise *Psi*, the Cholesky of the inverse with respect to the elimination scheme ordering.

25. begin for loop, $i = 1, \ldots, p$ to sample the diagonal entries of *Psi*.

26. begin internal if to test whether the parent set needs to be found. For the last case $i = p$, the vertex is the first in the perfect ordering so has no parents.

27. the pth diagonal element is the square root of a χ^2_δ.

28. begin alternative for $i < p$.

29. clear *nu_i*, the previous i number of parents.

30. need to calculate *nu_i*, the number of parents of the ith vertex in the perfect ordering. This is equal to the number of adjacent vertices which come after the ith vertex in the perfect elimination ordering. Vertices adjacent to i are represented by a 1 in the ith row. Those following are indicated by ones in the $i + 1, \ldots p$ columns of the ith row. Hence the number of parents, *nu_i*, is given by the sum of the elements of the ith row from the $i + 1$st to the pth column of the adjacency matrix *g_rev_perf*.

31. set the ith diagonal as the square root of a $\chi^2_{\delta+nu\_i}$ variable.

32. end if test for last vertex.

33. end for loop to sample the diagonal entries.

34. begin for loop, $i = 1, \ldots, p - 1$ down the rows to sample the off-diagonal entries.

35. begin for loop, $j = i + 1, \ldots, p$ across the columns in the upper diagonal half of *Psi*.

36. begin if test for sampling only the entries ij such that the edge $(i, j) \in$ *g_rev_perf*.

37. if $(i, j) \notin$ *g_rev_perf*, set the ijth entry to zero.

38. otherwise, for $(i, j) \in$ *g_rev_perf*, sample the ijth entry from a $N(0, 1)$ distribution.

39. end if test for edges

40. end j column loop.

41. end i row loop.

42. initialise *K_rev_perf*$\sim HW(g\_rev\_perf, \delta, I)$ and *sigma_id_rev_perf*$\sim HIW(g\_rev\_perf, \delta, I)$, the *g_rev_perf*-constrained hyper Wishart and hyper inverse Wishart matrices of the inverse covariance and covariance respectively.

43. calculate *K_rev_perf* as the 'square' *Psi'Psi*.

44. calculate *sigma_id_rev_perf* as the matrix inverse of *K_rev_perf*.

45. next need to 'undo' the permutation given by the perfect elimination reordering. begin by initialising *inverse_permute*, the vector of the inverse permutation that satisfies *inverse_permute*(*rev_perf*) $= I$, for I the identity permutation.

46. begin for loop, $j = 1, \ldots, p$ to create the inverse permutation vector.

47. the jth entry of the inverse permutation is given by the index of the vertex labelled j in *rev_perf*. This is because the original graph g is indexed as $1, \ldots, p$. So if the permutation of the perfect elimination is, for example, *rev_perf*$= (3, 1, 2, 4)$, then need to apply the permutation *inverse_permute*$= (2, 3, 1, 4)$ to get back the string $(1, 2, 3, 4)$.

48. end for loop to find the inverse permutation vector.

49. initialise *K_id*$\sim HW(g, \delta, I)$ and *sigma_id*$\sim HIW(g, \delta, I)$, the g-constrained hyper Wishart and hyper inverse Wishart matrices of the inverse covariance and covariance respectively.

50. calculate *K_id* by applying permutation *inverse_permute* to the rows and columns of *K_rev_perf*.

51. calculate *sigma_id* by applying permutation *inverse_permute* to the rows and columns of *sigma_rev_perf*.

```
function [sigma_id, K_id]=generate_HIW_g_delta_identity_cell(g, cliques, delta)
% inputs: 1. g, the p x p symmetric matrix with
%           respect to an original ordering v_1, ..., v_p
```

```
%         2. cliques, a 1 x t cell array of a perfect sequence of
%            (nonempty) cliques of g,
%            such as from chordal_to_ripcliques_cell.m
% output: 1. sigma_id, a random draw from HIW(g, delta, identity)
%   2. K_id=inv(sigma_id), a random draw from HW(g, delta, identity)

% THEORY: Roverato00 Theorem 3.
% This routine is first step in generating Sigma_i~HIW(g_i, delta, Phi_i).

   index_finish=0; index_start=0;
   num_cliques=size(cliques, 1);
   [seps, residuals, histories]=seps_resids_hists(cliques);
    % NOTE seps{1} will be [], because everyone writes them as S_2,...
   num_Rj=zeros(1, num_cliques);

%% create perfect ordering as per C1, R2, R3,..definition
   p=length(g);
   perfect_order=zeros(1,p);
   perfect_order(1:length(cliques{1}))=cliques{1};
   index_finish=length(cliques{1});
   for j=2:num_cliques
Rj=residuals{j};
num_Rj(1,j)=length(Rj);
index_start=index_finish+1;
index_finish=index_start+num_Rj(j)-1;
perfect_order(index_start:index_finish)=Rj;
   end

% now reverse the ordering, and use rev_perf=opposite
% order to perfect_order
% for indexing the graph and the parameter Phi
%% NOTE: can NOT assume that the ordering of g_reverse=
% my mcs ordering of g (had i computed it) in reverse. But
% by construction, it is opposite to a perfect order which
% satisfies at least one mcs with respect to g, and is constructed
% as C1, R2, R2, .., Rk as req'd by Roverato00p100

   rev_perf=zeros(1,p);
   index=0;
   for i=1:p
      rev_perf(i)=perfect_order(p-index);
      index=index+1;
   end

   g_rev_perf=zeros(p);
   g_rev_perf=g(rev_perf,rev_perf);

%%% to go back and forth: note that
% perf_order=p-rev_perf+1; perf+reverse=p+1
```

```
   Psi=zeros(p); % Psi is wrt reverse perfect ordering
% Psi(i,i) is a random Chi_squared(delta+nu_i).
   for i=1:p
      if i==p,
     Psi(p,p)=(chi2rnd(delta))^(.5);
      else
          %  find nu_i=num of parents=adj. predecessors
          % of each vertex i wrt PERFECT
          % order, NOT reverse= an elimination order. So nu_p=0
  % because Psi is wrt reverse ordering.
  % nu_i, i>1 could be zero for disconnected case.
         clear nu_i
         nu_i=sum(g_rev_perf(i,i+1:n));
         Psi(i,i)=(chi2rnd(delta+nu_i))^(.5);
      end
   end

%  Psi(i,j), j >i is N(0,1) if an edge
%  i,j exists in g, and zero otherwise.
   for i=1:p-1
      for j=i+1:p
         if (g_rev_perf(i,j)==0)
    Psi(i,j)=0;
 else
            Psi(i,j)=randn;
         end
      end
   end

   K_rev_perf=zeros(p); sigma_id_rev_perf=zeros(p);
   K_rev_perf=Psi'*Psi;
   sigma_id_rev_perf=inv(K_rev_perf);
      % is HIW(g_rev_perf, delta, id_rev_perf)
      % Note that this is covariance of g wrt the rev_perf ordering,
      % NOT wrt g. Should have zeros where off diagonal g_rev_perf does.

   %%% Now need to do inverse permutation to get back to the Sigma for g
   inverse_permute=zeros(1,p);
   for j=1:p
      inverse_permute(j)=find(rev_perf==j);
   end
   K_id=zeros(p); sigma_id=zeros(p);
   K_id=K_rev_perf(inverse_permute, inverse_permute);
   sigma_id=sigma_id_rev_perf(inverse_permute, inverse_permute); % ~HIW(g, delta, id)
% Check that sigma_id has zeros in right place:
% inv_sigma_id=inv(sigma_id);
% indices=find(~(inv_sigma_id==0)); inv_sigma_id(indices)=1;
% note that indices is not pairs, but where (1,1)=1, (1,2)=2, etc.

%%%%
```

```
% NOTES
% ------
% it is critical that the adjacency matrix g_rev_perf
% is indexed OPPOSITE to the node ordering given by
% C_1, ,,,, C_k a PERFECT order of g (NOT g_elim),
% i.e. that given by C_1, R_2, R_3,... etc. (Roverato00
% p 100 2.1 para3 lines 3-> based on those preceding.)
% DESPITE the fact that any perfect mcs ordering
% is satisfies the reverse being a perfect mcs
% ordering also, regardless of the size of the
% last clique, or the first node you choose of the
% last clique to begin. And any perfect sequence
% of cliques can be reversed and it's still a perfect
% sequence (so long as it was created under mcs)
% see my notes in folder "Generation: g must be wrt
% elim=perf1 opposite, but cliques with respect to
% perf1" and "Perfect=reverse Perfect" attached to "Perfect Orders".

% NOTE matlab orders union(a,b) as [min(a,b), max(a,b)]
      % regardless of relative size of a,b
      % so my cliques will always be ordered in strictly
      % ascending order [min,..max]
      % i.e. union([7,1,11,3], [4,9,18,2])
      % becomes [1,2,3,4,7,9,11,18]
```

The matrix array code is now explained.

1. initialise *index_finish* and *index_start*, the variables for indexing the sequence of vertices in *perfect_order*.

2. find *num_cliques*, the number of cliques. In the matrix array representation, $cliques(i, j) = 1$ if and only if the ith vertex is an element of the jth clique. Hence the number of cliques is given by the number of columns with at least one nonzero entry. This number can be found by finding the maximum index of the columns which have nonzero sum.

3. find *residuals*, the matrix array representation of the set of residuals.

4. initialise *num_Rj*, a vector in which the ith entry is the number of elements in the jth residual. Note that the sequence of (nonempty) residuals begins at $j = 2$.

5. find p, the number of vertices in g.

6. initialise *perfect_order*, the $1 \times p$ permutation vector of indicies $1, \ldots p$ ($1, \ldots p$ is the original indexing in the adjacency matrix g) that gives the perfect numbering which

when reversed is the permutation vector of a perfect elimination scheme. Note that entries in *perfect_order* must be nonzero integers, not one/zero indicators of vertices inclusion or not.

7. find *c_1*, the indicies of the vertices in the first clique with respect to the original ordering $1, \ldots, p$ of g. Note that the elements will be used to create a permutation, so the zero/one representation must be converted to an integer that is the vertex index in the graph. Hence *c_1* is given by the MATLAB find function on the first column of the matrix array representation *cliques*.

8. order vertices in C_1 first in the perfect numbering.

9. set *index_finish* as the index of the last element entered in the sequence so far. This is $|C_1|$. Then *index_start* can be updated to the last index plus one in the loop that follows.

10. begin for loop, $j = 2, \ldots,$ *num_cliques* to reorder the vertices.

11. set *Rj* as the jth residual. It must be a vector of integer indicies, not the zero/one representation in order to create the permutation vector. Hence it is given by the MATLAB find function on the jth column of the matrix array *residuals*.

12. find $|Rj|$, and store as the jth entry in *num_Rj*.

13. update *index_start* to the previous *index_finish* plus one.

14. update *index_finish*. This is the index of the last vertex entered in the current $|Rj|$. It is given by the current *index_start* plus $|Rj|$ less one.

15. enter R_j as the *index_start*th to *index_finish*th vertices in the perfect sequence.

16. end for loop for reordering vertices.

The remaining code is identical to the cell array version, so its itemised list description is omitted.

```
function [sigma_id, K_id, sigma_id_rev_perf, K_rev_perf]=generate_HIW_g_delta_identity_zo(g, cliques, delta)
% inputs: 1. g, the p x p symmetric matrix with
%            respect to an original ordering v_1, ..., v_p
%         2. cliques, a p x p matrix array representation of the cliques of g.
%            such as from chordal_to_ripcliques_zo.m
% output: 1. sigma_id, a random draw from HIW(g, delta, identity)
%   2. K_id=inv(sigma_id), a random draw from HW(g, delta, identity)
```

```
% THEORY: Roverato00 Theorem 3.
% GENERATE Sigma and inv(Sigma)=K, from g-conditional HIW(g, delta, identity):
% i.e. a Sigma for empty graph
% NOTE K_id has EXACT zeros in the 'right' places, no rounding errors
% First step to generate a random draw=Sigma_i~HIW(g_i, delta, Phi_i),
% where Phi_i are iterate
% outputs of the mcmc for each graph iterate g_i.

   index_finish=0; index_start=0;
   num_cliques=max(find( sum(cliques,1)~=0));
   [seps, residuals, histories]=seps_resids_hists_zo(cliques);
      % NOTE the seps(:,1) will be zeros(p,1), because everyone writes them as S_2,...
   num_Rj=zeros(1, num_cliques);
%%% create perfect ordering as per C1, R2, R3,..definition
   p=length(g);
   perfect_order=zeros(1,p);
   c_1=find(cliques(:,1))';
   perfect_order(1:length(c_1))=c_1;
   index_finish=length(c_1);
   for j=2:num_cliques
      Rj=find(residuals(:,j))'; % note the transpose
      num_Rj(1,j)=length(Rj);
      index_start=index_finish+1;
      index_finish=index_start+num_Rj(j)-1;
      perfect_order(index_start:index_finish)=Rj;
   end

% now reverse the ordering, and use rev_perf=opposite
% order to perfect_order
% for indexing the graph and the parameter Phi.

%% NOTE: can NOT assume that the ordering of g_reverse=
% my mcs ordering of g (had i computed it) in reverse. But
% by construction, it is opposite to a perfect order which
% satisfies at least one mcs with respect to g, and is constructed
% as C1, R2, R2, .., Rk as req'd by Roverato00p100

   rev_perf=zeros(1,p);
   index=0;
   for i=1:p
      rev_perf(i)=perfect_order(p-index);
      index=index+1;
   end

   g_rev_perf=zeros(p);
   g_rev_perf=g(rev_perf,rev_perf);

%%%% to go back and forth: note that
% perf_order=p-rev_perf+1; perf+reverse=p+1
```

```
%%%%%%%%%%

   Psi=zeros(p); % Psi is wrt reverse perfect ordering

% Psi(i,i) is a random Chi_squared(delta+nu_i).
   for i=1:p
      if i==p,
         Psi(p,p)=(chi2rnd(delta))^(.5);
      else
         % find nu_i=num of parents=adj. predecessors
         % of each vertex i wrt PERFECT
         % order, NOT reverse= an elimination order.
         % nu_p=0 because Psi is wrt reverse ordering.
         % nu_i, i>1 could be zero for disconnected case.
         clear nu_i
         nu_i=sum(g_rev_perf(i,i+1:p));
         Psi(i,i)=(chi2rnd(delta+nu_i))^(.5);
      end
   end

%  Psi(i,j), j >i is random N(0,1) if an edge
%  i,j exists in g, and zero otherwise.

   for i=1:p-1
      for j=i+1:p
         if (g_rev_perf(i,j)==0)
           Psi(i,j)=0;
        else
            Psi(i,j)=randn;
         end
      end
   end

   K_rev_perf=zeros(p); sigma_id_rev_perf=zeros(p);
   K_rev_perf=Psi'*Psi;
   sigma_id_rev_perf=inv(K_rev_perf);  %  is HIW(g_rev_perf, delta, id_rev_perf)
     % Note that this is covariance of g wrt the rev_perf ordering,
     % NOT wrt g. Should have zeros where g_rev_perf does, less diag

%%% Now need to do inverse permutation to get back to the Sigma for g
   inverse_permute=zeros(1,p);
   for j=1:p
      inverse_permute(j)=find(rev_perf==j);
   end
   K_id=zeros(p); sigma_id=zeros(p);
   K_id=K_rev_perf(inverse_permute, inverse_permute);
   sigma_id=sigma_id_rev_perf(inverse_permute, inverse_permute); % ~HIW(g, delta, id)
% Check that sigma_id has zeros in right place:
%inv_sigma_id=inv(sigma_id);
% indices=find(~(inv_sigma_id==0)); inv_sigma_id(indices)=1;
```

```
% note that indices is not pairs, but where (1,1)=1, (1,2)=2, etc.
```

8.1.16 closed transformation of $HIW(g, \delta, \bullet)$

The below code transforms $\Sigma_B \sim HIW(g, \delta, B)$ to $\Sigma_D \sim HIW(g, \delta, D)$ based on Theorem 4.3.3 and the theory explained in Section 4.3. It first creates the perfect elimination scheme by reversing the order given by $C_1, R_2, \ldots, R_k$. Next, it permutes the ordering of the adjacency matrix g to the perfect elimination order. Theorem 4.3.3 can then be used to calculate the inverse covariance $K_D^{rev\_perf} \sim HW(g\_rev\_perf, \delta, D)$ and $\Sigma_D^{rev\_perf} \sim HIW(g\_rev\_perf, \delta, D)$ which are indexed according to *g_rev_perf*, in which the vertices are enumerated according to a perfect elimination scheme. By Paulsen et al. (1989) the transformation is independent of the elimination scheme chosen, so $K_D \sim HW(g, \delta, D)$ and $\Sigma_D \sim HIW(g, \delta, D)$ are obtained by permuting the columns of $K_D^{rev\_perf}$ and $\Sigma_D^{rev\_perf}$ so that they are ordered as the original enumeration of the vertices in g.

The cell array version is described before the matrix array version. The itemised list descriptions are enumerated with respect to the lines of MATLAB code (excluding comment and blank lines) that immediately follow each.

1. find p, the number of vertices in g.

2. find *num_cliques*, the number of cliques.

3. find *seps, resids, hists*, the sets of vertex indicies representing the separators, residuals and histories, respectively.

4. initialise *num_Cj*, a vector in which the ith entry is the number of elements in the jth clique.

5. initialise *num_Sj*, a vector in which the ith entry is the number of elements in the jth separator. Note that the sequence of (nonempty) separators begins at $j = 2$, so *num_Sj(1,1)=0*.

6. initialise *num_Rj*, a vector in which the ith entry is the number of elements in the jth residual. Note that the sequence of (nonempty) residuals begins at $j = 2$, so *num_Rj(1,1)=0*.

7. find $|C_1|$, and enter as the first element of *num_Cj*.

8. initialise *perfect_order*, the $1 \times p$ permutation vector of indicies $1, \ldots p$ ($1, \ldots p$ is the original indexing in the adjacency matrix g) that gives the perfect numbering which when reversed is the permutation vector of a perfect elimination scheme.

9. order vertices in C_1 first in the perfect numbering.

10. set *index_finish* as the index of the last element entered in the sequence so far. This is $|C_1|$. Then *index_start* can be updated to the last index plus one in the loop that follows.

11. begin for loop, $j = 2, \ldots,$ *num_cliques* to reorder the vertices.

12. set *Cj*,*Rj* and *Sj* as the index representation of the jth clique, residual and separator respectively.

13. find $|C_j|$, $|R_j|$, $|S_j|$, and store as the jth entry in *num_Cj*, *num_Rj* and *num_Sj* respectively.

14. update *index_start* to the previous *index_finish* plus one.

15. update *index_finish*. This is the index of the last vertex entered in the current $|Rj|$. It is given by the current *index_start* plus $|Rj|$ less one.

16. enter R_j as the *index_start*th to *index_finish*th vertices in the perfect sequence.

17. end for loop for reordering vertices.

18. initialise *rev_perf*, the $1 \times p$ permutation vector that gives the reverse perfect numbering required for a perfect elimination scheme.

19. initialise *index* to zero. This is a counter for putting the $p - index$th element of *perfect_order* at the ith position in *rev_perf*.

20. begin for loop, $i = 1, \ldots, p$ to create *rev_perf*.

21. the ith entry in the reverse seqence is the $p - index$th entry in *perfect_order*.

22. increment *index* by one.

23. end for loop to create *rev_perf*.

24. apply the permutation *rev_perf* to the rows and columns of g to give *g_rev_perf*. To undo the permutation, the ith entry of perfect order permutation is related to the jth entry in its reverse by the relation $i = p - j + 1$. That is, $i + j = p + 1$. For example, for the $p = 5$ sequence enumerations, we have $[1, 2, 3, 4, 5] + [5, 4, 3, 2, 1] = 6 = (p+1)$.

25. apply the permutation *rev_perf* to the rows and columns of B to give *B_rev_perf*, the permuted version of B. B is the parameter of the HIW distributed matrix Σ_B that is to be transformed.

26. apply the permutation *rev_perf* to the rows and columns of D to give *D_rev_perf*. D is the parameter of the HIW distributed matrix Σ_D which is the result of transforming $\Sigma_B \sim HIW(g, \delta, B)$ to $\Sigma_D \sim HIW(g, \delta, D)$.

27. apply the permutation *rev_perf* to the rows and columns of the input matrix *sigma_B* D to give *sigma_B_rev_perf*.

28. calculate *K_B*, the inverse of *sigma_B_rev_perf*.

29. calculate *choleskyK_B*, the Cholesky of *K_B*.

30. set *c1* as the number of vertices in C_1.

31. clear previous values of *index_start* and *index_finish*

32. set *index_start* to $p - |C_1| + 1$ and *index_finish*$= p$.

33. initialise a $1 \times |C_1$ variable *indexC1_in_Upsilon_D*. This variable will be the indicies of C_1 in the perfect elimination scheme; i.e. the indicies of the last $|C_1|$ columns of the *Upsilon_D*. *Upsilon_D* is the Cholesky of the inverse of *sigma_D* with respect to the ordering of the perfect elimination scheme.

34. set *indexC1_in_Upsilon_D*$= [p - |C_1| + 1, cdots, p]$.

35. initialise *Upsilon_D*, the Cholesky of the inverse matrix.

36. initialise the $|C_1 \times C_1|$ matrices *B_1* and *D_1*. These will be the C_1 dependent subblocks of B and D respectively.

37. Similarly initialise *Q_1*, *P_1* and *O_1* of Theorem 4.3.3.

38. find *B_1* as the last $|C_1|$ row/column subblock of the permuted *B_rev_perf* (as per Theorem 4.3.3).

39. calculate *Q_1*, the Cholesky of the inverse of *B_1* (as per Theorem 4.3.3).

40. find *D_1* as the last $|C_1|$ row/column subblock of the permuted *D_rev_perf* (as per Theorem 4.3.3).

41. calculate *P_1*, the Cholesky of the inverse of *D_1* (as per Theorem 4.3.3).

42. calculate $O\_1 = (Q\_1)^{-1} P\_1$ (as per Theorem 4.3.3).

43. calculate the C_1 dependent subblock of *Upsilon_D* (as per Theorem 4.3.3).

44. begin *for* loop, $j = 2, \ldots$ *num_cliques* to calculate *Upsilon_D*.

45. clear previous values of *cj* and *indexCj_in_Upsilon_D*. These are the j dependent variables analogous to *c1* and *indexC1_in_Upsilon_D*.

46. find $|C_j|$.

47. initialise the $1 \times |C_j|$ vector *indexCj_in_Upsilon_D*.

48. clear previous values of *Rj*, *rj* *indexRj_inOj* and *indexRj_in_Upsilon_D*. *Rj* is the jth residual R_j, and *rj*= $|R_j|$. *indexRj_inOj* is the vector of indicies of R_j in the submatrix *O_j*. *O_j* has columns and rows enumerated to follow the perfect elimination scheme. *indexRj_in_Upsilon_D* is defined analogously to *indexCj_in_Upsilon_D*.

49. clear previous *unsort_indexRj_in_Upsilon_D*. the kth entry in this vector is the index in the perfect elimination order, of the kth vertex in R_j, where R_j is with respect to the original enumeration of vertices in g. For example, if the $k = 3$rd vertex in R_j is $v_8 \in g = (V, E)$, and the perfect elimination permutation is $[4, 1, 5, 6, 3, 8, \ldots,]$ then *unsort_indexRj_in_Upsilon_D*$(3) = 6$. The prefix *unsort* indicates that the entries in *unsort_indexRj_in_Upsilon_D* are not sorted from minimum to maximum or vice versa.

50. find R_j.

51. calculate *rj*= $|R_j|$.

52. initialise *indexRj_inOj*.

53. initialise *indexRj_in_Upsilon_D*.

54. initialise *unsort_indexRj_in_Upsilon_D*.

55. clear previous values of *Sj, sj, indexSj_inOj* and *indexSj_in_Upsilon_D*. They are each defined analogously to the respective R_j dependent variable.

56. clear previous *unsort_indexSj_in_Upsilon_D*, which is defined analogously to *unsort_indexRj_in_Upsilon_D*

57. find S_j.

58. calculate $sj = |S_j|$.

59. initialise *indexSj_inOj*.

60. initialise *indexSj_in_Upsilon_D*.

61. initialise *unsort_indexSj_in_Upsilon_D*.

62. initialise the $|C_j \times C_j|$ matrices *B_j* and *D_j*. These will be the C_j dependent subblocks of B and D respectively.

63. Similarly initialise *Q_j*, *P_j* and *O_j* of Theorem 4.3.3.

64. begin inner for loop, $k = 1, \ldots rj$ to calculate *unsort_indexRj_in_Upsilon_D*.

65. set the kth entry of *unsort_indexRj_in_Upsilon_D* as the position in the perfect elimination permutation *rev_perf* of the kth vertex in R_j.

66. end inner for loop, $k = 1, \ldots rj$ to calculate *unsort_indexRj_in_Upsilon_D*.

67. sort the elements of *unsort_indexRj_in_Upsilon_D* in ascending order to give the position indexes in *Upsilon_D* of the vertices in R_j. This step is not essential, as MATLAB takes subblocks $A_(B, B)$ of any matrix B independently of the ordering of the position indexes in B.

68. begin inner for loop, $k = 1, \ldots sj$ to calculate *unsort_indexSj_in_Upsilon_D*.

69. set the kth entry of *unsort_indexSj_in_Upsilon_D* as the position in the perfect elimination permutation *rev_perf* of the kth vertex in S_j.

70. end inner for loop, $k = 1, \ldots sj$ to calculate *unsort_indexSj_in_Upsilon_D*.

71. sort the elements of *unsort_indexSj_in_Upsilon_D* in ascending order to give the position indexes in *Upsilon_D* of the vertices in S_j.

72. Because the sequences of separators and residuals are each enumerated with respect to the perfect sequence of cliques, and because the sequence of separators includes repetitions, the cliques $C_j = S_j \cup R_j$. Hence can calculate the position indexes of *indexCj_in_Upsilon_D* as the position indexes in *indexRj_in_Upsilon_D* followed by those in *indexSj_in_Upsilon_D*. Note that R_j follows S_j as *Upsilon_D* is enumerated in a reverse perfect numbering.

73. find *B_j* as the C_j row/column subblock of the permuted *B_rev_perf* (as per Theorem 4.3.3).

74. calculate *Q_j*, the Cholesky of the inverse of *B_j* (as per Theorem 4.3.3).

75. find *D_j* as the C_j row/column subblock of the permuted *D_rev_perf* (as per Theorem 4.3.3).

76. calculate *P_j*, the Cholesky of the inverse of *D_j* (as per Theorem 4.3.3).

77. calculate $O\_j = (Q\_j)^{-1} P\_j$ (as per Theorem 4.3.3).

78. set the position indexes of C_j as $1, \ldots, |C_j|$. The position indexes of R_j in this subblock are $1, \ldots |R_j|$. The positon indexes of S_j follow these, so are given by $|R_j| + 1, \ldots |C_j|$

79. calculate the R_j, R_j dependent subblock of *Upsilon_D* (as per Theorem 4.3.3).

80. calculate the R_j, S_j dependent subblock of *Upsilon_D* (as per Theorem 4.3.3).

81. end external *for* loop to calculate *Upsilon_D*.

82. initialise *K_D_rev_perf* and *sigma_D_rev_perf*. These are the transformed inverse covariance and covariance, respectively, with rows and columns enumerated as per the perfect elimination ordering.

83. calculate $K\_D\_rev\_perf = Upsilon\_D' Upsilon\_D$.

84. calculate $sigma\_D\_rev\_perf = (K\_D\_rev\_perf)^{-1}$.

85. calculate *inverse_permute*, the inverse permutation vector which will order the variables as per the original vertex enumeration in g.

86. find the inverse covariance $K\_D \sim HW(g, \delta, D)$ by applying the inverse permutation to the rows and columns of *K_D_rev_perf*.

87. finally find the transformed covariance $sigma\_D \sim HIW(g, \delta, D)$ by applying the inverse permutation to the rows and columns of *sigma_D_rev_perf*.

```
function [sigma_D, K_D]=transform_g_conditional_HIW_cell(sigma_B, g, cliques, delta, B, D)
% inputs: 1. sigma_B~HIW(g, delta, B), the p x p covariance
%         2. g, the p x p symmetric matrix with
%            respect to an original ordering v_1, ..., v_p
%         3. cliques, a 1 x t cell array of a perfect sequence of
%            (nonempty) cliques of g,
%            such as from chordal_to_ripcliques_cell.m
%         4. delta, the degrees of freedom parameter of the distribution of sigma_B
%         5. B, the matrix parameter of the distribution of sigma_B
%         6. D, the matrix parameter of the distribution of the transformed covariance
% output: 1. sigma_D, a random draw from HIW(g, delta, D)
%         2. K_D=inv(sigma_D), a random draw from HW(g, delta, D)

% THEORY: Roverato00 Theorem 4.
% sigma_B~g-conditional HIW(g, delta, B)
% to sigma_D~g-conditional HIW(g, delta, D)
% K_D is inv sigma_D
% NOTE B and D constrained to satisfy inv(B)(i,j)=0 iff g(i,j)=0.
% use to generate a random draw=Sigma_i~HIW(g_i, delta, Phi_i),
% where Phi_i are iterate
% outputs of the mcmc for each graph iterate g_i.

   p=length(g);
   num_cliques=length(cliques);
   [seps, residuals, histories]=seps_resids_hists(cliques);
      % NOTE the seps{1} will be [], because everyone writes them as S_2,...
   num_Cj=zeros(1, num_cliques);
   num_Sj=zeros(1, num_cliques); %num_Sj(1,1)=num_Rj(1,1)=0;
   num_Rj=zeros(1, num_cliques);
   num_Cj(1,1)=length(cliques{1});

%%% create perfect ordering as per C1, R2, R3,..definition
   perfect_order=zeros(1,p);
   perfect_order(1:length(cliques{1}))=cliques{1};
   index_finish=length(cliques{1});
   for j=2:num_cliques
      Cj=cliques{j}; Rj=residuals{j}; Sj=seps{j};
      num_Cj(1,j)=length(Cj);num_Rj(1,j)=length(Rj);num_Sj(1,j)=length(Sj);
      index_start=index_finish+1;
      index_finish=index_start+num_Rj(j)-1;
      perfect_order(index_start:index_finish)=Rj;
   end

% now reverse the ordering, and use rev_perf=opposite
% order to perfect_order
% for indexing the graph and the parameter Phi
%% NOTE: can NOT assume that the ordering of g_reverse=
```

```
% my mcs ordering of g (had i computed it) in reverse. But
% by construction, it is opposite to a perfect order which
% satisfies at least one mcs with respect to g, and is constructed
% as C1, R2, R2, .., Rk as req'd by Roverato00p100

   rev_perf=zeros(1,p);
   index=0;
   for i=1:p
      rev_perf(i)=perfect_order(p-index);
      index=index+1;
   end
   g_rev_perf=g(rev_perf,rev_perf);
   B_rev_perf=B(rev_perf,rev_perf);
   D_rev_perf=D(rev_perf,rev_perf);
%%%% to go back and forth: note that
% perf_order=p-rev_perf+1; perf+reverse=p+1

%%%% transformation as per thm 4 roverato2000
% sigma_B~HIW(g_rev_perf, delta, B_rev_perf) to
% Sigma_D~HIW(g_rev_perf, delta, D_rev_perf)
% Let Matrix_cj be the sub matrix of Matrix indexed by cliques of
% g NOT g_elim, and similarly for the separators and residuals.

   sigma_B_rev_perf=sigma_B(rev_perf,rev_perf);
   K_B=inv(sigma_B_rev_perf);
   choleskyK_B=chol(K_B);

   c1=num_Cj(1,1);
   clear index_start index_finish
   index_start=p-c1+1; index_finish=p;
   indexC1_in_Upsilon_D=zeros(1,c1);
   indexC1_in_Upsilon_D=[index_start: index_finish];
      % this is the last c1 columns

   Upsilon_D=zeros(p);
   B_1=zeros(c1); D_1=zeros(c1);
   Q_1=zeros(c1); P_1=zeros(c1); O_1=zeros(c1);

   B_1=B_rev_perf(indexC1_in_Upsilon_D,indexC1_in_Upsilon_D);
   Q_1=chol(inv(B_1));
   D_1=D_rev_perf(indexC1_in_Upsilon_D,indexC1_in_Upsilon_D);
   P_1=chol(inv(D_1));
   O_1=inv(Q_1)*P_1;    % this is a letter O
   Upsilon_D(indexC1_in_Upsilon_D, indexC1_in_Upsilon_D)=...
             choleskyK_B(indexC1_in_Upsilon_D,indexC1_in_Upsilon_D)*O_1;

   for j=2:num_cliques
      clear cj indexCj_in_Upsilon_D
      cj=num_Cj(j);
      indexCj_in_Upsilon_D=zeros(1, cj);
```

```
   clear Rj rj indexRj_inOj indexRj_in_Upsilon_D
   clear unsort_indexRj_in_Upsilon_D
   Rj=residuals{j};
   rj=num_Rj(j);
   indexRj_inOj=zeros(1,rj);
   indexRj_in_Upsilon_D=zeros(1,rj);
   unsort_indexRj_in_Upsilon_D=zeros(1,rj);

   clear Sj sj indexSj_inOj indexSj_in_Upsilon_D
   clear unsort_indexSj_in_Upsilon_D
   Sj=seps{j};
   sj=num_Sj(j);
   indexSj_inOj=zeros(1,sj);
   indexSj_in_Upsilon_D=zeros(1,sj);
   unsort_indexSj_in_Upsilon_D=zeros(1,sj);

   B_j=zeros(cj);  D_j=zeros(cj);
   Q_j=zeros(cj);  P_j=zeros(cj); O_j=zeros(cj); % this is a letter O

   for k=1:rj
      unsort_indexRj_in_Upsilon_D(k)=find(rev_perf==Rj(k));
   end
   indexRj_in_Upsilon_D=sort(unsort_indexRj_in_Upsilon_D);

   for k=1:sj
      unsort_indexSj_in_Upsilon_D(k)=find(rev_perf==Sj(k));
   end
   indexSj_in_Upsilon_D=sort(unsort_indexSj_in_Upsilon_D);

   indexCj_in_Upsilon_D=[indexRj_in_Upsilon_D, indexSj_in_Upsilon_D];

   B_j=B_rev_perf(indexCj_in_Upsilon_D,indexCj_in_Upsilon_D);
   Q_j=chol(inv(B_j));
   D_j=D_rev_perf(indexCj_in_Upsilon_D,indexCj_in_Upsilon_D);
   P_j=chol(inv(D_j));
   O_j=inv(Q_j)*P_j;

   indexRj_inOj=1:rj;
   indexSj_inOj=rj+1:cj;

   Upsilon_D(indexRj_in_Upsilon_D, indexRj_in_Upsilon_D)=...
                            choleskyK_B(indexRj_in_Upsilon_D, indexRj_in_Upsilon_D)*...
                            O_j(indexRj_inOj, indexRj_inOj);
   Upsilon_D(indexRj_in_Upsilon_D,indexSj_in_Upsilon_D)=...
                            choleskyK_B(indexRj_in_Upsilon_D,indexRj_in_Upsilon_D)*...
                            O_j(indexRj_inOj,indexSj_inOj)+...
                            choleskyK_B(indexRj_in_Upsilon_D,indexSj_in_Upsilon_D)*...
                            O_j(indexSj_inOj,indexSj_inOj);
end
```

```
   K_D_rev_perf=zeros(p);  sigma_D_rev_perf=zeros(p);
   K_D_rev_perf=Upsilon_D'*Upsilon_D;    % ~HW(g_rev_perf, delta, D_rev_perf)
   sigma_D_rev_perf=inv(K_D_rev_perf);  % ~HIW(g_rev_perf, delta, D_rev_perf)
      % Note that this is covariance of g wrt the rev_perf ordering, NOT wrt g.
%%% Now need to do inverse permutation to get back to the Sigma for g
   inverse_permute=zeros(1,p);
   for j=1:p
      inverse_permute(j)=find(rev_perf==j);
   end
   K_D=zeros(p);
   sigma_D=zeros(p);
   K_D=K_D_rev_perf(inverse_permute, inverse_permute); % ~Wishart(g, delta+p-1, D) (E(K_D) prop inv(D))
   sigma_D=sigma_D_rev_perf(inverse_permute, inverse_permute); % ~HIW(g, delta, D) (E(D) prop D
```

The matrix array version differs in only two points. Firstly, the number of cliques is given by: `num_cliques=max(find( sum(cliques,1)~=0));`. The ijth entry of the $p \times p$ matrix cliques is one if and only if the ith vertex is in the jth clique, so the number of cliques is equal to the index of the last column that has nonzero sum. Secondly, the extra lines which convert a zero/one column representation of a set of vertices to an integer string representation which consists of the vertex indicies with respect to the original enumeration $1, \ldots, p$ in g. The conversion is achieved by applying the MATLAB find function to the appropriate column of the approprate matrix array, as follows:

```
Cj=find(cliques(:,j))'; % note the transpose
Rj=find(residuals(:,j))'; % note the transpose
Sj=find(seps(:,j))'; % note the transpose
```

The line by line description is omitted.

```
function [sigma_D, K_D, sigma_D_rev_perf, K_D_rev_perf]=...
              transform_g_conditional_HIW_no_mcs_zo(sigma_B, g, cliques, delta, B, D)
% inputs: 1. sigma_B~HIW(g, delta, B), the p x p covariance
%         2. g, the p x p symmetric matrix with
%            respect to an original ordering v_1, ..., v_p
%         3. cliques, a p x p matrix array representatino of a perfect sequence of
%            cliques of g,
%            such as from chordal_to_ripcliques_zo.m
%         4. delta, the degrees of freedom parameter of the distribution of sigma_B
%         5. B, the matrix parameter of the distribution of sigma_B
%         6. D, the matrix parameter of the distribution of the transformed covariance
% output: 1. sigma_D, a random draw from HIW(g, delta, D)
%         2. K_D=inv(sigma_D), a random draw from HW(g, delta, D)

% THEORY: Roverato00 Theorem 4.
% sigma_B~g-conditional HIW(g, delta, B)
% to sigma_D~g-conditional HIW(g, delta, D)
% K_D is inv sigma_D
% NOTE B and D constrained to satisfy inv(B)(i,j)=0 iff g(i,j)=0.
```

```
% use to generate a random draw=Sigma_i~HIW(g_i, delta, Phi_i),
% where Phi_i are iterate
% outputs of the mcmc for each graph iterate g_i.

   p=size(g,1);
   num_cliques=max(find( sum(cliques,1)~=0));
   [seps, residuals, histories]=seps_resids_hists_zo(cliques);
      % NOTE the seps{1} will be [], because everyone writes them as S_2,...
   num_Cj=zeros(1, num_cliques);
   num_Sj=zeros(1, num_cliques); %num_Sj(1,1)=num_Rj(1,1)=0;
   num_Rj=zeros(1, num_cliques);
   num_Cj(1,1)=sum(cliques(:,1));

%%% create perfect ordering as per C1, R2, R3,..definition
   perfect_order=zeros(1,p);
   perfect_order(1:num_Cj(1,1))=find(cliques(:,1))';
   index_finish=num_Cj(1,1)
   for j=2:num_cliques
      Cj=find(cliques(:,j))'; % note the transpose
      Rj=find(residuals(:,j))'; % note the transpose
      Sj=find(seps(:,j))'; % note the transpose
      num_Cj(1,j)=length(Cj);
      num_Rj(1,j)=length(Rj);
      num_Sj(1,j)=length(Sj);

      index_start=index_finish+1;
      index_finish=index_start+num_Rj(j)-1;
      perfect_order(index_start:index_finish)=Rj;
   end

% now reverse the ordering, and use rev_perf=opposite
% order to perfect_order
   rev_perf=zeros(1,p);
   index=0;
   for i=1:p
      rev_perf(i)=perfect_order(p-index);
      index=index+1;
   end

   g_rev_perf=g(rev_perf,rev_perf);
   B_rev_perf=B(rev_perf,rev_perf); % generate wrt new order
   D_rev_perf=D(rev_perf,rev_perf);
%%%% to go back and forth: note that
% perf_order=p-rev_perf+1; perf+reverse=p+1

%%%% transformation as per thm 4 roverato2000
% sigma_B~HIW(g_rev_perf, delta, B_rev_perf) to
% Sigma_D~HIW(g_rev_perf, delta, D_rev_perf)
%%Let Matrix_cj be the sub matrix of Matrix indexed by cliques of
%g NOT g_elim, and similarly for the separators and residuals.
```

```
sigma_B_rev_perf=sigma_B(rev_perf,rev_perf); % SIGMA in roverato2000
K_B=inv(sigma_B_rev_perf); % inv(SIGMA)PHI in rov
choleskyK_B=chol(K_B);  %PHI in rov

c1=num_Cj(1,1);
clear index_start index_finish
index_start=p-c1+1; index_finish=p;
indexC1_in_Upsilon_D=zeros(1,c1);
indexC1_in_Upsilon_D=[index_start: index_finish]; % this is the last c1 columns

Upsilon_D=zeros(p);
B_1=zeros(c1);
D_1=zeros(c1);
Q_1=zeros(c1);
P_1=zeros(c1);
O_1=zeros(c1);

B_1=B_rev_perf(indexC1_in_Upsilon_D,indexC1_in_Upsilon_D);
Q_1=chol(inv(B_1));
D_1=D_rev_perf(indexC1_in_Upsilon_D,indexC1_in_Upsilon_D);
P_1=chol(inv(D_1));
O_1=inv(Q_1)*P_1;    % this is a letter O
Upsilon_D(indexC1_in_Upsilon_D, indexC1_in_Upsilon_D)=...
        choleskyK_B(indexC1_in_Upsilon_D,indexC1_in_Upsilon_D)*O_1;

for j=2:num_cliques
   clear cj indexCj_in_Upsilon_D
   cj=num_Cj(j);
   indexCj_in_Upsilon_D=zeros(1, cj);

   clear Rj rj indexRj_inOj
   Rj=find(residuals(:,j))'; %Rj=residuals{j};
   rj=num_Rj(j);
   indexRj_in_Upsilon_D=zeros(1,rj);
   indexRj_inOj=zeros(1,rj);
   unsort_indexRj_in_Upsilon_D=zeros(1,rj);

   clear Sj sj indexSj_inOj
   Sj=find(seps(:,j))'; %=seps{j};
   sj=num_Sj(j);
   unsort_indexSj_in_Upsilon_D=zeros(1,sj);
   indexSj_in_Upsilon_D=zeros(1,sj);
   indexSj_inOj=zeros(1,sj);
   B_j=zeros(cj);
   D_j=zeros(cj);
   Q_j=zeros(cj);
   P_j=zeros(cj);
   O_j=zeros(cj); % this is a letter O
```

```
      for k=1:rj
         unsort_indexRj_in_Upsilon_D(k)=find(rev_perf==Rj(k));
      end
      indexRj_in_Upsilon_D=sort(unsort_indexRj_in_Upsilon_D);

      for k=1:sj
       unsort_indexSj_in_Upsilon_D(k)=find(rev_perf==Sj(k));
      end
      indexSj_in_Upsilon_D=sort(unsort_indexSj_in_Upsilon_D);

      indexCj_in_Upsilon_D=[indexRj_in_Upsilon_D, indexSj_in_Upsilon_D];

      B_j=B_rev_perf(indexCj_in_Upsilon_D,indexCj_in_Upsilon_D);
      Q_j=chol(inv(B_j)); % this is trivially I(Cj,Cj)
      D_j=D_rev_perf(indexCj_in_Upsilon_D,indexCj_in_Upsilon_D);
      P_j=chol(inv(D_j));
      O_j=inv(Q_j)*P_j;

      indexRj_inOj=1:rj;
      indexSj_inOj=rj+1:cj;

      Upsilon_D(indexRj_in_Upsilon_D, indexRj_in_Upsilon_D)= choleskyK_B(indexRj_in_Upsilon_D, indexRj_in_Upsilon_D)*...
                          O_j(indexRj_inOj, indexRj_inOj);
      Upsilon_D(indexRj_in_Upsilon_D,indexSj_in_Upsilon_D)= choleskyK_B(indexRj_in_Upsilon_D,indexRj_in_Upsilon_D)*...
                          O_j(indexRj_inOj,indexSj_inOj)+...
                          choleskyK_B(indexRj_in_Upsilon_D,indexSj_in_Upsilon_D)* O_j(indexSj_inOj,indexSj_inOj);
   end
   K_D_rev_perf=zeros(p);
   sigma_D_rev_perf=zeros(p);
   K_D_rev_perf=Upsilon_D'*Upsilon_D;
   sigma_D_rev_perf=inv(K_D_rev_perf);  % ~HIW(g_rev_perf, delta, D_rev_perf)
      % Note that this is covariance of g wrt the rev_perf ordering, NOT wrt g.

%%% Now need to do inverse permutation to get back to the Sigma for g
   inverse_permute=zeros(1,p);
   for j=1:p
      inverse_permute(j)=find(rev_perf==j);
   end
   K_D=zeros(p);
   sigma_D=zeros(p);
   K_D=K_D_rev_perf(inverse_permute, inverse_permute); % ~Wishart(g, delta+p-1, D)
   sigma_D=sigma_D_rev_perf(inverse_permute, inverse_permute); % ~HIW(g, delta, D)
```

8.1.17 calculating the logarithm of the normalising constant for the hyper inverse Wishart distribution.

The below code calculates the normalising constant $h(g, \delta, \Phi)$ of (4.5) derived in Section 4.2. The code is self explanatory, apart from the call to subroutine *mvt_gamma_ln.m* which is given in Subsection 8.1.24. Therefore the line by line description is omitted. The cell array

version is given first, followed by the matrix array version. The matrix array version is identical apart from the representation of the cliques and separator sets. Note that a test for $S_i = \emptyset, i > 1$ is necessary. In the case of a disconnected graph, there will be at least 2 empty separators based on the definition $S_i = C_i \cap H_{i-1}$. These will be for each i such that C_i is the top of a connected junction tree componenet in a junction forest representation of g.

```
function [ln_h]=h_constant_ln_cell(cliques, delta, Phi)
% inputs: 1. cliques, a 1 x t cell array of the t nonempty
%            cliques of g in RIP ordering
%            (from chordal_to_ripcliques_cell.m)
%         2. delta, integer > 0
%         3. Phi>0, p x p parameter for Sigma_E~HIW(g, delta, Phi)
% output: 1. log of h(g, delta, Phi), the normalising constant for
%            p(Sigma_E) where E is set of edges in g

% Appears in h_likelihood, the
% g-constrained likelihood p(Y=y |g).
% Y~ p-di Normal(0, inv(Omega))
% Based on Roverato 2000, Prop 2

   ln_prod_top_terms=0;
   ln_prod_bottom_terms=0;
   seps=seps_resids_hists_cell(cliques);

   for i=1:length(cliques)
      C_i=cliques{i};
      Phi_C_i=Phi(C_i, C_i);
      numC_i=length(C_i);
      ln_top_term_i= ( (delta+ numC_i -1) /2 ) * log(det(Phi_C_i/2) )...
         - mvt_gamma_ln( numC_i, (delta+ numC_i -1) /2);

      ln_prod_top_terms=ln_prod_top_terms+ln_top_term_i;

      if i==1,
         ln_bottom_term_i=0;
      elseif isempty(seps{i});
         % need this case for disconnected graph.
         % If there are 2 components, then 2 empty seps
          ln_bottom_term_i=0;
      else
         S_i=seps{i};
         numS_i=length(S_i);
         Phi_S_i=Phi(S_i, S_i);
         ln_bottom_term_i=( (delta+ numS_i -1) /2 )* log(det(Phi_S_i/2))...
            - mvt_gamma_ln( numS_i, (delta+ numS_i -1) /2);
      end
      ln_prod_bottom_terms=ln_prod_bottom_terms+ln_bottom_term_i;
   end

   ln_h=ln_prod_top_terms-ln_prod_bottom_terms;
```

```
% recall det(cA)=c^p det(A) where A is nxn


function [ln_h]=h_constant_ln_zo(cliques, delta, Phi)
% inputs: 1. cliques, a p x p matrix repesentation of the
%            cliques of g in RIP ordering
%            (from chordal_to_ripcliques_zo.m)
%         2. delta, integer > 0
%         3. Phi>0, p x p parameter for Sigma_E~HIW(g, delta, Phi)
% output: 1. log of h(g, delta, Phi), the normalising constant for
%            p(Sigma_E) where E is set of edges in g

% Appears in h_likelihood, the
% g-constrained likelihood p(Y=y |g).
% Y~ p-di Normal(0, inv(Omega))
% Based on Roverato 2000, Prop 2

   ln_prod_top_terms=0;
   ln_prod_bottom_terms=0;
   seps=seps_resids_hists_zo(cliques)

   for i=1:length(cliques)
      C_i=cliques(:,i);
      C_i_nodes=find(C_i);
      Phi_C_i=Phi(C_i_nodes, C_i_nodes);
      numC_i=sum(C_i);
      ln_top_term_i= ( (delta+ numC_i -1) /2 ) * log(det(Phi_C_i/2) )...
         - mvt_gamma_ln( numC_i, (delta+ numC_i -1) /2);

      ln_prod_top_terms=ln_prod_top_terms+ln_top_term_i;

      if i==1,
         ln_bottom_term_i=0;
      elseif sum(seps(:,i))==0;
         % need this case for disconnected graph.
         % If there are 2 components, then 2 empty seps

         ln_bottom_term_i=0;
      else
         S_i_nodes=seps(:,i);
         numS_i=length(S_i_nodes);
         Phi_S_i=Phi(S_i_nodes, S_i_nodes);
         ln_bottom_term_i=( (delta+ numS_i -1) /2 )* log(det(Phi_S_i/2))...
         - mvt_gamma_ln( numS_i, (delta+ numS_i -1) /2);
      end
      ln_prod_bottom_terms=ln_prod_bottom_terms+ln_bottom_term_i;
   end

   ln_h=ln_prod_top_terms-ln_prod_bottom_terms;
% recall det(cA)=c^n det(A) where A is nxn
```

8.1.18 calculating the logarithm of the ratio of normalising constants $h(g, \delta, \Phi)/h(g', \delta, \Phi)$.

Let $e = (u, v) \in E$ be an edge in $g = (V, E)$ but not in $g' = (V, E')$. Section 4.4 shows that the ratio of marginal likelihoods $p(y|g^p, \delta, \Phi)/p(y|g^c, \delta, Phi)$ (4.18) in the MH acceptance probability for the transition proposal (4.17) required for sampling from the posterior is a function of $h(g, \delta, \Phi)/h(g', \delta, \Phi)$. This ratio can be calculated using (4.8) of Lemma 4.4.3, and the terms in (4.8) can be calculated efficiently using Lemma 4.4.5. The routines *ln_h_ratio_cell.m* and *ln_h_ratio_zo.m* calculate $h(g, \delta, \Phi)/h(g', \delta, \Phi)$ using the results of Theorem 4.4.1, Lemma 4.4.3 and Lemma 4.4.5. The code is self explanatory so the line by line description is omitted. The cell array version is given first, followed by the matrix array version. The matrix array version is identical apart from the representation of the new clique C.

```
function [ln_h]=ln_h_ratio_cell( C, a, b, delta, Phi)
% inputs: 1. C, the new clique
%            as represented by a 1 x |C| array of vertex indicies
%            with respect to original g.
%            In the case of edge addition C
%            contains the vertices of the added edge.
%            In the case of edge deletion, C contains the deleted edge.
%         2. a, b, the index of the edge vertices with respect to the
%            original ordering in the adjacency matrix g.
%         3. delta, integer > 0
%         4. Phi>0, p x p parameter for Sigma_E~HIW(g, delta, Phi)
% output: 1. log of h(g, delta, Phi)/h(g, delta, Phi), the
%            ratio of normalising constants for
%            p(Sigma_E) where E is set of edges in g, and
%            p(Sigma_E') where E is set of edges in g'.

%  marginal likelihood ratio is  h_ratio(delta, Phi)/h_ratio(delta+n, Phi+S_N)
% (the 2*pi terms cancel out)

   D=[a,b];
   Sq2=mysetdiff(C,D);
   numSq2=length(Sq2);
   delta_star=delta -1;
   Phi_CC=zeros(length(C));
   Phi_CC=[ Phi(Sq2, Sq2) Phi(Sq2,D) ;...
               Phi(D, Sq2)    Phi(D,D)];
   L=(chol(Phi_CC))' ;
   L_DD=L([numSq2+1, numSq2+2], [numSq2+1, numSq2+2]);
      % this is correct: DD is the LOWER block.
      % Sq2 is index of upper block.
      % Hence the terms of the matrix containing DD
      % are indexed as numSq2+1, numSq2+2
   l_aa=L_DD(1,1);
   l_bb=L_DD(2,2);
   l_ab=L_DD(2,1); % L is lower diag: L_DD(1,2)=0
   ln_top=(delta_star + numSq2 +2)*(log(l_aa )+log( l_bb));
```

```
   ln_bottom= (delta_star + numSq2 +1)/2*...
                        (log(l_aa^2*l_ab^2 + l_aa^2*l_bb^2)) ;
   ln_gam_bit=-log(2*sqrt(pi))+gammaln(( delta_star + numSq2 +1)/2 )...
                                        - gammaln(( delta_star + numSq2 +2)/2 );
   ln_h= ln_top-ln_bottom + ln_gam_bit;
```

```
function [ln_h]=ln_h_ratio_zo( C, a, b, delta, Phi)
% inputs: 1. C, the p x 1 zero/one vector representation of the new clique
%           which in the case of edge addition
%           contains the vertices of the added edge, or
%           in the case of edge deletion, contained the deleted edge.
%         2. a, b, the index of the edge vertices with respect to the
%            original ordering in the adjacency matrix g.
%         3. delta, integer > 0
%         4. Phi>0, p x p parameter for Sigma_E~HIW(g, delta, Phi)
% output: 1. log of h(g, delta, Phi)/h(g, delta, Phi), the
%            ratio of normalising constants for
%            p(Sigma_E) where E is set of edges in g, and
%            p(Sigma_E') where E is set of edges in g'.

% marginal likelihood ratio is  h_ratio(delta, Phi)/h_ratio(delta+n, Phi+S_N)
% (the 2*pi terms cancel out)
```

```
   C_nodes=(find(C))'; % NOTE THE TRANSPOSE
   D=[a,b];
       % C_nodes and D are the actual nodes, not
       % the col_vec representation
   Sq2=setdiff(C_nodes,D);
   numSq2=length(Sq2);
   delta_star=delta -1;
   Phi_CC=zeros(length(C_nodes));
   Phi_CC=[ Phi(Sq2, Sq2) Phi(Sq2,D) ;...
                Phi(D, Sq2)    Phi(D,D)];
   L=(chol(Phi_CC))' ;
   L_DD=L([numSq2+1, numSq2+2], [numSq2+1, numSq2+2]);
      % this is correct: DD is the LOWER block.
      % Sq2 is index of upper block.
      % Hence the terms of the matrix containing DD
      % are indexed as numSq2+1, numSq2+2
   l_aa=L_DD(1,1);
   l_bb=L_DD(2,2);
   l_ab=L_DD(2,1); % L is lower diag: L_DD(1,2)=0
   ln_top=(delta_star + numSq2 +2)*(log(l_aa )+log( l_bb));
    ln_bottom= (delta_star + numSq2 +1)/2*...
                        (log(l_aa^2*l_ab^2 + l_aa^2*l_bb^2)) ;
   ln_gam_bit=-log(2*sqrt(pi))+gammaln(( delta_star + numSq2 +1)/2 )...
                                        - gammaln(( delta_star + numSq2 +2)/2 );
   ln_h= ln_top-ln_bottom + ln_gam_bit;
```

8.1.19 randomly selecting a pair of vertices.

An irreducible chain of decomposable graphs results from single legal edge additions and deletions. Rather than generating the edge indicators e_{ij} one at a time, an alternative and efficient methodology is to randomly select a vertex pair (i, j) and use a Metropolis Hastings proposal for deleting the edge $e = (i, j)$ if the pair are adjacent, or adding the edge e if they are not. The proposal is also conditional on the legality of the edge change. This routine is used to randomly propose a pair of vertices (a, b). If the pair of vertices are adjacent in the current chain iterate g^c, and $e = (a, b)$ is a legal deletion, then this edge will be proposed for deletion. If the pair are not adjacent and e is a legal addition, then e is proposed for addition. Note that the pair will only constitute a proposal if the resulting change is legal Also note that the proposal graph g^p will only be accepted as the next chain iterate if it passes the Metropolis Hastings proposal test.

The following 4 lines of description are enumerated with respect to the 4 lines of MATLAB code (excluding comment and blank lines) which follow them.

1. find p, the number of vertices.

2. define *iota* to be a random permutation of the vertex indicies $1, \ldots, p$.

3. choose the first and second vertices as the output. Note that the choice of the first and second is arbitrary. Since iota is a random permutation, any pair will do.

4. define the 1×2 array *edge_candidate* to be the representation of $e = (v_i, v_j)$.

```
function [edge_candidate]=next_edge_candidate(g_current)
% input: 1. g_current, the p x p symmetric adjacency matrix
%           of the current iterate graph of the chain with
%           respect to an original ordering v_1, ..., v_p
% output: 1. edge_candidate, the randomly chosen vertices which
%            will comprise the edge to be changed to give the
%            proposal graph in the MHMCMC sampler IF the resulting
%            graph is decomposable.

% g_proposal will only be
% accepted in that routine as the new g_current if it passes the acceptance
% ratio
   p=length(g_current);
   iota = randperm(p);   % iota is a random permutation of 1:p.
   i=iota(1); j = iota(2);
   edge_candidate=[i,j];
```

8.1.20 proposing the next graph

An irreducible chain of decomposable graphs results from single legal edge additions and deletions. By choosing a distribution on the graph space, a Metropolis Hastings or Gibbs reduced conditional sampler can be used to transverse the space. The routine *next_graph_candidate.m* generates a proposal graph for consideration in a Metropolis Hastings proposal based on such a measure. It creates the proposal graph g^p from the current graph g^c as follows. First randomly select a pair of vertices v_i, v_j. If $v_i \nsim v_j$ in g^c and $e = (v_i, v_j)$ is a legal addition, then create g^p from g^c by adding e. If $v_i \sim v_j$ in g^c and $e = (v_i, v_j)$ is a legal deletion, then create g^p from g^c by deleting e. Note that the proposal graph does not depend on a distribution, apart from the uniform measure on the decomposable graph space. Also note that g^p will only be accepted as the next chain iterate if it passes the Metropolis Hastings proposal test.

The code for the matrix array version is identical to that of the cell array version excepting that it makes calls to functions which assume the matrix representation of sets. Therefore, only the itemised list description of the cell array version is given. It is enumerated with respect to the lines of MATLAB code (excluding comment and blank lines) which follow it.

1. find p, the number of vertices.
2. initialise the adjacency matrix *g_proposal* of g^p as the adjacency matrix *g_current* of the current graph iterate $g^c = (V, E^c)$.
3. initialise to 0 *CASE_add* and *CASE_delete*, the 0(no)/1(yes) case indicators for adding or deleting an edge from g^c, respectively.
4. begin *while* loop for continuing the random proposal of edge vertex pairs to test.
5. while no legal edge change has been made to g^c, use *next_edge_candidate.m* to propose a pair of vertices, or equivalently, an edge candidate. The edge candidate is represented by the 1×2 vector *edge*, with $edge(1)$ the index of the first vertex of the pair, and $edge(2)$ the second.
6. set the edge $e = (v_{edge(1)}, v_{edge(2)})$.
7. begin if condition for legal deletion of e.
8. if $e \in E^c$ (so the ijth entry of the adjacency matrix *g_current* is one), then call the function *check_edge_delete.m* to see if deleting e from g^c is legal.

9. begin if test to see if the proposed deletion was legal.

10. if e is a legal deletion, then delete it; i.e. set the respective entries of the adjacency matrix $g\_proposal(edge(1), edge(2)) = 0$.

11. if e is a legal deletion, then set *CASE_delete*= 1 and define C as the output of *check_edge_delete.m*. This represents the clique in g^c that contained e. The calculation of the likelihood depends on this clique, so it must be outputted by this routine.Set a and b as the indicies of the vertices constituting e.

12. end internal if test for legal deletion.

13. begin alternative $elseif$ for $v_i \nsim v_j$, and not connected.

14. if v_i and v_j are in different connected components of g^c, then the edge addition is legal, so add the set *CASE_add*= 1, $e = (v_i, v_j)$ and $C = \{v_i, v_j\}$.

15. if v_i and v_j are in different connected components of g^c, then add the legal edge to g^c to give the proposal graph g^p.

16. begin alternative $elseif$ test for $v_i \nsim v_j$, but connected.

17. if $v_i \nsim v_j$ but connected, then call the program *check_edge_add_same_component.m* to see if $e = (v_i, v_j)$ is a legal addition.

18. begin internal if test based on the output indicator of the program *check_edge_add_same_component.m*. If the indicator is 1, then adding e is legal.

19. if adding e is legal, set the output edge vertices of the extra edge to be v_i, v_j and set C as the clique indicies that were outputted by *check_edge_add.m*. This clique is needed to calculate the likelihood, so must be outputted by *next_graph_candidate.m*.

20. if e is a legal addition, create the proposal graph by adding e to g^c.

21. end internal if test for the legality of adding e to g^c.

22. end conditional $elseif$ case for $v_i \nsim v_j$, but connected.

23. end external while loop.

```
function [g_proposal, a,b, C, CASE_add, CASE_delete]=...
   next_graph_candidate_cell(g_current, jtree, sepsize, cliques, reach_graph)
% inputs: 1. g_current, the adjacency matrix of the current graph in the chain.
%         2. jtree, the adjacency matrix of a junction tree with respect
```

```
%         to cliques.
%         3. sepsize, a matrix array of the size of the separator sets, in which
%         sepsize(i,j) = [number of elements in intersection between
%         cliques cliques{i} and cliques{j} if they are adjacent, and zero else.
%         4. cliques, a 1 x t cell array of nonempty cliques
%         of a chordal graph in RIP ordering
%         (such as re_index_cliques from chordal_to_ripcliques_cell.m).
%         5. reach_graph, a p x p symmetric matrix, in which
%         reach_graph(i,j) = 1 iff there is a path from v_i to v_j in g
% outputs:1. g_proposal, the adjacency matrix of the graph proposed for
%         the next graph in the chain. g_proposal will only be
%         accepted as g_next if it passes the MH acceptance test.
%         2., 3. a, b, the indicies of the vertices of
%         the edge whose addition or deletion from g_current gives g_proposal.
%         4. C, the clique in g_proposal which contains the extra edge
%         in the case of addition, or the clique which contains the edge
%         in g_current which is proposed for deletion.
%         Likelihood depends on C, so it is outputted by this routine.
%         5., 6. CASE_add, CASE_delete, the 0/1 indicators of whether
%         an edge is proposed for deletion or addition respectively.
   p=length(g_current);
   g_proposal=g_current;
      % initialise the proposal graph to the current graph
   CASE_add=0; CASE_delete=0;
   while isequal(g_proposal,g_current);
      edge=next_edge_candidate(g_current);
      i=edge(1); j=edge(2);
      if g_current(i,j)==1,
      [CASE_delete, C_potential_delete]=check_edge_delete_cell(i,j, cliques);
         if ( CASE_delete==1) ;
            g_proposal(i,j)=0; g_proposal(j,i)=0;
            CASE_delete=1; a=i; b=j; C=C_potential_delete;
         end
      elseif ( g_current(i,j)==0 & reach_graph(i,j)==0 );
         CASE_add=1; a=i; b=j; C=[a,b];
            % if adding edge between 2 disjoint trees, new clique
            % including that edge must be comprised soley of edge nodes a, b
         g_proposal(i,j)=1; g_proposal(j,i)=1;
            % if v_i and v_j are in different connected components of a decomposable
            % graph, then adding (v_i, v_j) is legal.
      elseif ( g_current(i,j)==0 & reach_graph(i,j) ==1 ),
        % v_i, v_j must be in the same connected component of tree for check_edge_add.
         [CASE_add, C_potential_add]=...
            check_edge_add_same_component_cell(i,j, jtree, sepsize, cliques);
         if CASE_add==1
           a=i; b=j; C=C_potential_add;
           g_proposal(i,j)=1; g_proposal(j,i)=1;
         end
      end
   end;  % external while
```

```
function [g_proposal, a,b, C, CASE_add, CASE_delete]=...
   next_graph_candidate_zo(g_current, jtree, sepsize, cliques, reach_graph)
% inputs: 1. g_current, the adjacency matrix of the current graph in the chain.
%         2. jtree, the adjacency matrix of a junction tree with respect
%         to cliques.
%         3. sepsize, a matrix array of the size of the separator sets, in which
%         sepsize(i,j) = [number of elements in intersection between
%         cliques cliques{i} and cliques{j} if they are adjacent, and zero else.
%         4. cliques, a p x p matrix representation of the cliques
%         of a chordal graph in RIP ordering
%         (such as cliques from chordal_to_ripcliques_zo.m).
%         5. reach_graph, a p x p symmetric matrix, in which
%         reach_graph(i,j) = 1 iff there is a path from v_i to v_j in g
% outputs:1. g_proposal, the adjacency matrix of the graph proposed for
%         the next graph in the chain. g_proposal will only be
%         accepted as g_next if it passes the MH acceptance test.
%         2., 3. a, b, the indicies of the vertices of
%         the edge whose addition or deletion from g_current gives g_proposal.
%         4. C, the p x 1 representation of the
%         clique in g_proposal which contains the extra edge
%         in the case of addition, or the clique which contains the edge
%         in g_current which is proposed for deletion.
%         Likelihood depends on C, so it is outputted by this routine.
%         5., 6. CASE_add, CASE_delete, the 0/1 indicators of whether
%         an edge is proposed for deletion or addition respectively.
   p=length(g_current);
   g_proposal=g_current;
      % initialise the proposal graph to the current graph
   CASE_add=0; CASE_delete=0;
   while isequal(g_proposal,g_current);
      edge=next_edge_candidate(g_current);
      i=edge(1); j=edge(2);
      if g_current(i,j)==1,
         [CASE_delete, C_potential_delete]=check_edge_delete_zo(i,j, cliques);
         if (CASE_delete==1) ;
            g_proposal(i,j)=0; g_proposal(j,i)=0;
            CASE_delete=1; a=i; b=j; C=C_potential_delete;
         end
      elseif ( g_current(i,j)==0 & reach_graph(i,j)==0 );
         CASE_add=1; a=i; b=j; C=[a,b];
      % if adding edge between 2 disjoint trees, new clique
      % including that edge must be comprised soley of edge nodes a, b
         g_proposal(i,j)=1; g_proposal(j,i)=1;
         % if v_i and v_j are in different connected components of a decomposable
         % graph, then (v_i,v_j) is a legal addition.
      elseif ( g_current(i,j)==0 & reach_graph(i,j) ==1 ),
         % v_i,v_j must be in the same connected component of tree for check_edge_add.
         [CASE_add, C_potential_add]=...
         check_edge_add_same_component_zo(i,j, jtree, sepsize, cliques);
         if CASE_add==1
```

```
          % if check_edge_add_same_component(i,j, jtree, sepsize, cliques)==1,
          a=i; b=j; C=C_potential_add;
          g_proposal(i,j)=1; g_proposal(j,i)=1;  CASE2=1;
        end
      end;
    end; % external while
```

8.1.21 sampling the next graph iterate

This function can be used on its own as a reduced conditional sampler for g. Alternatively, by omitting all the analysis outputs, and setting the number of iterations equal to 1, it can be used to generate the next graph iterate in a covariance selection sampler such as *generate_regression_covariance_RWMH.m* which is given in Subsection 8.1.22.

Note that *generate_graph_and_analysis_cell.m* uses the Giudici & Green (1999) junction tree characterisation for legal edge additions. It therefore calculates the junction tree and uses junction tree dependent subroutines. *generate_graph_and_analysis_cell.m* makes calls to most of the programs already given in this appendix.

The only section that requires explanation is:

```
   if CASE_add==1,
      ln_likelihood_ratio=ln_h_ratio_cell( C, a, b, delta, Phi)...
            -ln_h_ratio_cell( C, a, b, delta+N, Phi+S_N);
      likelihood_ratio=exp(ln_likelihood_ratio);
   elseif CASE_delete==1,
      ln_likelihood_ratio=ln_h_ratio_cell( C, a, b, delta+N, Phi+S_N)...
            -ln_h_ratio_cell( C, a, b, delta, Phi);
      likelihood_ratio=exp(ln_likelihood_ratio);
   end
```

This section of code calculates the natural logarithm of the graph marginal likelihood ratio in (4.18) using Lemma 4.4.3.

The rest of the code is self-explanatory, so a line by line explication is omitted in favour of comments within the code which follows.

```
function [g_next, graph_pm, graph_tally, graph_tally_counts, graph_iter]=...
   generate_graph_and_analysis_cell(g_current, delta, Phi, N, S_N, warmup, iter_multiple, iters_to_keep)
% inputs: 1. g_current, adj. matrix of current graph iterate
%         2., 3. delta, Phi, the parameters of the HIW(g, delta, Phi) prior for the covariance
%         3. N, the number of observations
%         4. S_N, the sum of squares matrix of the data
%         5. warmup, the number of warmup iterations
%         6. iter_multiple, the factor of iters_to_keep for the total number of iterates
%         7. iters_to_keep, the number of graph iterates to record
% outputs: 1. g_next, the last graph iterate
```

```
%          2. graph_pm, a p x p adjacency matrix of the average graph sampled
%          3. graph_tally, a pxpx num_diff_graphs array of the distinct
%             adjacency matrices.
%          3. graph_tally_counts, a 1 x p vector of counts of each graph sampled.
p=length(Phi);
iter=iter_multiple*iters_to_keep;
if (iter <= warmup),
  error('not enough iterations specified ') ;
end
sample=iter-warmup;
counter_for_keeping=0;
graph_cumulative=zeros(p,p);
graph_pm=zeros(p,p);
graph_iter=zeros(n,n, iters_to_keep);
graph_tally=zeros(p,p, 1);
graph_tally_counts=zeros(p,p,1);
num_of_diff_graphs=1;

accept_count_warmup=0; accept_count_sample=0;
accept_percent_warmup=0; accept_percent_sample=0;

g_next=g_current;
for i=1:iter;
   clear cliques jtree order clear sepsize reach_graph
   clear g_proposal g_i
   equal_indicator=0;

   g_current=g_next;
   cliques=chordal_to_ripcliques_cell(g_current) ; % find cliques
   jtree=ripcliques_to_jtree_cell(cliques);  % create jtree
   sepsize=separators_cell(cliques, jtree); % find size of separators
   reach_graph=reachability_graph_cell(g_current); % find path matrix

   u=rand; accept_prob=0;
   [g_proposal, a,b, C, CASE_add, CASE_delete]=...
      next_graph_candidate_cell(g_current, jtree, sepsize, cliques, reach_graph);

   if CASE_add==1,
      ln_likelihood_ratio=ln_h_ratio_cell( C, a, b, delta, Phi)...
            -ln_h_ratio_cell( C, a, b, delta+N, Phi+S_N);
      likelihood_ratio=exp(ln_likelihood_ratio);
   elseif CASE_delete==1,
      ln_likelihood_ratio=ln_h_ratio_cell( C, a, b, delta+N, Phi+S_N)...
            -ln_h_ratio_cell( C, a, b, delta, Phi);
      likelihood_ratio=exp(ln_likelihood_ratio);
   end

   accept_prob=min(1, likelihood_ratio);

   % accept g_proposal with probability=accept.
```

```
   if u<= accept_prob,
      g_next=g_proposal;
      if i<=warmup
        accept_count_warmup=accept_count_warmup+1;
      else accept_count_sample=accept_count_sample+1;
      end
   else g_next=g_current;
   end

   g_i=g_next;
      % store graph_i into graph_iter
   if mod(i, iter_multiple)==0
      counter_for_keeping=counter_for_keeping+1;
      graph_iter(:,:,counter_for_keeping)=g_next;
   end

   if i>warmup,
      graph_cumulative=graph_cumulative+g_i;
       % if g_i has already been sampled, update its count accordingly.
       % otherwise, add g_i to the array of distinct graphs
      for j=1:num_of_diff_graphs
         graph_tally_j=graph_tally(:,:,j);
         if (isequal(graph_tally_j, g_i));
            graph_tally_cumulative(:,:,j)=graph_tally_cumulative(:,:,j)+g_i;
            graph_tally_counts(1,j)=graph_tally_counts(1,j)+1;
            equal_indicator=1;
         break,
         end
      end  % end of for j=1: num_of_diff_graphs loop

      if equal_indicator==0
         num_of_diff_graphs=num_of_diff_graphs+1;
         graph_tally(:,:,num_of_diff_graphs)=g_i;
         graph_tally_cumulative(:,:, num_of_diff_graphs)=g_i;
         graph_tally_cumulative_counts(1,num_of_diff_graphs)=1;
      end
   end % end for the if i> warmup loop
end   % end for the i=1:iter
accept_percent_warmup=accept_count_warmup/warmup;
accept_percent_sample=accept_count_sample/sample;
graph_pm=graph_cumulative/sample;
```

The matrix array version is identical, excepting that it makes calls to the matrix representation alternatives of each subroutine. It is therefore omitted.

8.1.22 generating the covariance selection iterates

This function generates the next iterates in the covariance selection MCMC sampler described in Section 4.8.

A line by line explication of the code is omitted as each step is explained using comments in the program which follows. Only the matrix array version is given, as it is identical to the cell array version excepting calls to the respective cell array versions of each subroutine.

```
function [omega, sigma, g, order, cliques, tau, rho, Phi]=...
             generate_regression_covariance_RWMH_zo...
             (g, order, cliques, delta, tau, rho, Phi, N, S_N,...
               g_prior, An, ...
               gen_g,  sample_graph, warmup_graph,...
               model_Phi, var_ln_tau, var_rho)
% inputs: 1. g, adj. matrix of current graph iterate
%         2. order, a permutation vector giving a perfect numbering of vertices with
%            respect to the adjacency matrix g
%         3. cliques, a p x p matrix representation of a perfect sequence of cliques of g
%         4. delta, the degrees of freedom parameter for the prior on Sigma ~HIW(g, delta, Phi)
%         5., 6. tau, rho, the parameters in Phi. These are generated using a random walk
%            metropolis hastings scheme.
%         7. Phi, the matrix parameter of the HIW(g, delta, Phi) prior for the covariance
%         8. N, the number of observations
%         9. S_N, the sum of squares matrix of the data
%         10. g_prior, an indicator for whether to use the size or uniform prior
%         11. An, a vector such that An(k) is the number of graphs of size k on n vertices
%         12. gen_g, an indicator for the whether or not to thin the graph iterates
%         13., 14. sample_graph, warmup_graph, the number of sampling and warmup iterations
%             to use if thinning the graph iterates.
%         15. model_Phi, a string indicating which model of Phi is assumed. Possible values
%             are 'equicorrelated', 'scaled_SSY' and 'tauI'.
%         16., 17.  var_ln_tau, var_rho, the variances for ln(tau) and rho in the
%             random walk Metropolis Hastings sampler for tau and rho respectively.
% outputs: 1., 2. omega~HW(g, delta, Phi) and sigma~HIW(g, delta, Phi), the next
%            iterates of the inverse covariance and covariance, respectively.
%          3. g, the next graph iterate
%          4. order, a permutation vector giving a perfect numbering of vertices with
%             respect to the adjacency matrix of new g
%          5. cliques, a p x p matrix representation of a perfect sequence of cliques of new g
%          6., 7. tau, rho, the next iterates of the parameters in Phi.
%          8. Phi, the matrix parameter of the HIW(g, delta, Phi) prior for the covariance
   p=length(g);
    if (strcmp(gen_g,'single')==1),
        g_next=generate_graph_single_zo(g, g_prior, An, cliques, delta, Phi, N, S_N);
    elseif (strcmp(gen_g,'not_single')==1),
        g_next=generate_graph_zo(g, g_prior, An, delta, Phi, N, S_N, warmup_graph, sample_graph);
    else error_gen_g='no gen_g specified'
    end

    if ~isequal(g,g_next) % don't bother re-doing cliques if they ARE equal
```

```
      g=g_next;
      [check, order]=check_chordal(g_next);
      cliques=chordal_to_ripcliques_zo(g_next, order);
    end

% Generate next tau for current rho and graph:
%  routine depends on model_Phi internally, but
%  does not refer to rho unless equicorrelated model_Phi
   tau= generate_tau_RWMH_zo(tau, var_ln_tau, rho, model_Phi,...
                                        cliques, delta, N, S_N);

% Generate next rho for new tau and current graph IF equicorrelated model_Phi
  if (strcmp(model_Phi,'equicorrelated')==1)
    rho=generate_rho_RWMH_zo(rho, var_rho, tau, cliques, delta, N, S_N);
  end

% Update Phi (deterministically
  if (strcmp(model_Phi,'equicorrelated')==1)
     Phi=tau*(rho*ones(p)+(1-rho)*eye(p));
  elseif (strcmp(model_Phi,'scaled_SSY')==1),
     Phi=tau*S_N/N;
  elseif (strcmp(model_Phi,'tauI')==1),
     Phi=tau*eye(p);
  else error_Phi='no such model_Phi'
  end

% Generate sigma_next by deterministically transforming a randomly drawn
% HIW(g, delta+N, Identity) to HIW(g, delta+N, Post_Phi)
    sigma=zeros(p, p); Post_Phi=zeros(p, p); Post_Phi_hat=zeros(p, p); sigma_identity=zeros(p, p);
    Post_Phi=Phi+S_N;
    Post_Phi_hat=g_constrain_zo(Post_Phi, order, cliques);
    sigma_identity=generate_HIW_g_delta_identity_zo(g, cliques, delta+N);
    [sigma, omega, sigma_rev_perf, omega_rev_perf]=transform_g_conditional_HIW_zo...
           (sigma_identity, g, cliques, delta+N, eye(p), Post_Phi_hat);
```

8.1.23 decomposable covariance selection script.

This script was used to generate the results of covariance selection reported in the simulation sections, and is included to illustrate how the programs fit together for covariance selection. It assumes the size prior for the graphs, and the equicorrelated form of the parameter Φ. It is for $p = 17$, and uses 20 sets of data, each consisting of 40 observations. The sampler uses 20K iterations.

```
%% main_id17n40_eqsize_idXnrep20s20
%% volratio is neq*(neq-1)/2 long vector "scales" sigma.

% CURRENT 10:25 21/09/04 and works perfectly
%% this version includes nx_KL=1 case
%  this version uses RWMH for tau and rho
% _zo copied from CURRENT 09:25 8/04/04
```

```
% copied from helen_main 15:32 03/04/04
% copied from helen_main 12.46 23/3/04, then updated phat, KL bits only

%contains the following subroutines of ed's.
%init_CORR_J_R_Tvec_Jbind.m; initialise paramters
%comp_XtX_XtY_YtY.m; effecient computation of matrices outside loop
%comp_XtOX_XtOY_YtOY.m; effecient computation of matrices inside loop
%genb_post.m; generate Jbind and beta_vector
%comp_SSY.m efficient computation of SSY
%*************************************************************************

clear all
rand('state',sum(100*clock)) % causes matlab 5 generator to be used in rand calls
randn('state',sum(100*clock)) % causes matlab 5 generator to be used in randn calls
global dimb c_par maskb iota % ed's global parameters

load DATAFILESn17nobs40nrep20idX/data_id17n40nrep20idX

neq, nobs, model_sigma
dimb, c_par, 'maskb=', maskb', iota

% idX means that there are no covariate regressors in the
% X matrix: it is a matrix of constants, hence Xbeta=mean which is
% independent of any random variables; i.e. regressors
% i.e. no regressors in the model. Just estimating mean and covariance.

% in this case may as well set X as the identity matrix, so that Xbeta=beta
% and beta is therefore the estimate of the mean.

% this is the 'trivial' regression model.

load volratio_ful.dat
vol=volratio_ful(:,2);

clear warmup sampl
warmup=2000% 1000
sampl=20000% 10000% 5000

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% specify parameters and initial values for graph routines and sigma prior

model_Phi='equicorrelated', %'scaled_SSY', %'tauI', % 'equicorrelated', %  'tauI', %  'scaled_SSY', %  'tauI' 'equicorrelated' '
g_prior='size_dependent', %'uniform',%  'uniform',%   'size_dependent',% 'uniform',%

model_mean='mean_unknown' % ='mean_zero'%
gen_g='single';
sample_graph=1; warmup_graph=0;
delta=5;

% FINE TUNE BELOW so that acceptance rate is around 25%
```

```
%
   rho=1/2; % choose so that rho in (-1/(n-1), 1);
   tau=4;
   var_ln_tau=1/10;
   var_rho=1/20;  % this really gets into tails eg: rho <0 and rho >.9
         % recall 95% data within 2 std (i.e. p(-1.96sigma<E(Z)<1.96sigma)=.95
         % want variance so that generating from mini normal in (-1/(n-1), 1)

if (strcmp(model_mean,'mean_zero')==1)
          nobs_adjusted=nobs
elseif (strcmp(model_mean,'mean_unknown')==1)
          nobs_adjusted=nobs-1
           % need to adjust degrees of freedom by -1 as mu integrated out
end

nC2=neq*(neq-1)/2;
An=zeros(1, nC2+1);

%load Ank_fortran_estimates/A8k_true
%An=A8k_true(neq+1,1:nC2+1);

%load Ank_fortran_estimates/A15Kalls20.out
%An=A15Kalls20(1:nC2+1);

%load Ank_fortran_estimates/A16Kalls20.out
%An=A16Kalls20(1:nC2+1);

load Ank_fortran_estimates/A17Kalls20.out
An=A17Kalls20(1:nC2+1);

g=zeros(neq); % should i should initialise to my g equivalent of ed's Jind=upper only?
              % i.e. a decomposable version
              % NOTE: the empty graph corresponds to all independent.
[check, order]=check_chordal(g);
cliques=chordal_to_ripcliques_no_mcs_zo(g, order);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Jind_iter=zeros(neq,neq,warmup+sampl); % my g_iter
omega_iter=zeros(neq,neq,sampl);
sigma_iter=zeros(neq,neq,sampl);
Jbind_iter=zeros(dimb,sampl);
beta_vector_iter=zeros(dimb,sampl);
Jind_pm=zeros(neq,neq,nrep);
sigma_pm=zeros(neq,neq,nrep);
omega_pm=zeros(neq,neq,nrep);
Jbind_pm=zeros(dimb,nrep);
beta_vector_pm=zeros(dimb,nrep);

tau_iter=zeros(warmup+sampl,1);
```

```
tau_pm=zeros(1,nrep);
if (strcmp(model_Phi,'equicorrelated')==1)
  rho_iter=zeros(warmup+sampl,1);
  rho_pm=zeros(1,nrep);
end

if(SIMULATE)
   KL=zeros(1,nrep);
   L1=zeros(1,nrep);
   L1_SSY=zeros(1,nrep);
end %if simulate

fprintf(' all data are inputted \n')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% begin reps for KL

time_taken=zeros(1,nrep); % vector of times for reps
elapsed_time_per_mod=zeros(3, 5);  % just save 3 nrep's worth, and the first 5K
                    %zeros(nrep, 5);
for irep=1:nrep,
  count_mod=0;
  disp(sprintf('***REPLICATION: %d',irep))

  Y_irep=squeeze(Y_learn(irep,:,:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% initialise graph and eds stuff

g=zeros(neq); % should i should initialise to my g equivalent of ed's Jind=upper only?
              % i.e. a decomposable version
              % NOTE: the empty graph corresponds to all independent.
[check, order]=check_chordal(g);
cliques=chordal_to_ripcliques_no_mcs_zo(g, order);

if (strcmp(model_mean,'mean_zero')==1)
   SSY=Y_irep'*Y_irep;
elseif (strcmp(model_mean,'mean_unknown')==1)
   [omega,CORR,R,Jind,Tvec,Jbind]=init_CORR_J_R_Tvec_Jbind;
   content=[1:1:neq];
   [XtX_int,XtY_int,YtY_int]=comp_XtX_XtY_YtY(X,Y_irep);
   [XtOX,XtOY,YtOY]=comp_XtOX_XtOY_YtOY(XtX_int,XtY_int,YtY_int,omega);
   [Jbind,beta_vector]=genb_post(Jbind,YtOY,XtOX,XtOY);
   SSY=cov(Y_irep)*(nobs-1);
       % matlab covariance is sample average adjusted SSY/(nobs-1);
end

if (strcmp(model_Phi,'equicorrelated')==1)
   Phi=tau*(rho*ones(neq)+(1-rho)*eye(neq));
elseif (strcmp(model_Phi,'scaled_SSY')==1),
```

```
   Phi=tau*SSY/nobs_adjusted;
elseif (strcmp(model_Phi,'tauI')==1),
   Phi=tau*eye(neq);
else error_Phi='no such model_Phi'
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% begin mcmc loop

tic

phat_running_total=zeros(nx_KL, ny_KL); % initialise the jth sum phat matrix to zeros

for iter=1:warmup+sampl,

   if mod(iter,1000)==0  % print out replication every 1000, so can see where up to
     disp(sprintf('***ITERATION: %d',iter))
   end % if mod...
%
%% update Jbind and beta_vector.
%% specify constant matrices according to current values of TCORRT=omega.

     [XtOX,XtOY,YtOY]=comp_XtOX_XtOY_YtOY(XtX_int,XtY_int,YtY_int,omega);
     [Jbind,beta_vector]=genb_post(Jbind,YtOY,XtOX,XtOY);

%
% Generate sigma and omega

    [omega, sigma, g, order, cliques, tau, rho, Phi]=...
             generate_regression_covariance_RWMH_zo...
             (g, order, cliques, delta, tau, rho, Phi,...
               nobs_adjusted, SSY,...
               g_prior, An, ...
               gen_g,  sample_graph, warmup_graph,...
               model_Phi, var_ln_tau, var_rho);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     if (irep == 1)
        Jind_iter(:,:,iter)=g;
        tau_iter(iter,1)=tau;
        if (strcmp(model_Phi,'equicorrelated')==1)
          rho_iter(iter,1)=rho;
        end
     end

   if (iter > warmup)
       if (irep == 1)
         omega_iter(:,:,iter-warmup)=omega;
         sigma_iter(:,:,iter-warmup)=sigma;
         beta_vector_iter(:,iter-warmup)=beta_vector;
```

```
        Jbind_iter(:,iter-warmup)=Jbind;
      end %if irep==1.

      omega_pm(:,:,irep)=omega_pm(:,:,irep)+omega;
      sigma_pm(:,:,irep)=sigma_pm(:,:,irep)+sigma;
      beta_vector_pm(:,irep)=beta_vector_pm(:,irep) + beta_vector;
      Jbind_pm(:,irep)=Jbind_pm(:,irep)+Jbind;
      Jind_pm(:,:,irep)=Jind_pm(:,:,irep)+g; %my g_pm
      tau_pm(:,irep)=tau_pm(:,irep)+tau;
      if (strcmp(model_Phi,'equicorrelated')==1)
         rho_pm(:,irep)=rho_pm(:,irep)+rho;
      end

     if ~isequal(size(Y_KL), [nx_KL, ny_KL, neq])
        error_dim_KL='mistake in dimensions Y_KL'
     end

     if ~isequal(size(X_KL), [nx_KL, neq, dimb])
        error_dim_XL=' mistake in dimensions X_KL'
     end

     phat_running_total=phat_running_total+phat_irep_iter(X_KL,Y_KL,beta_vector,sigma, omega);

   end %if iter>warmup

   if mod(iter,1000)==0
     count_mod=count_mod+1;
     if irep<=3
      elapsed_time_per_mod(irep, count_mod)=toc
     end
   end %end if mod...

end  %  end MCMC loop

  time_taken(irep)=toc

%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      omega_pm(:,:,irep)=omega_pm(:,:,irep)/sampl;
      sigma_pm(:,:,irep)=sigma_pm(:,:,irep)/sampl;
      beta_vector_pm(:,irep)=beta_vector_pm(:,irep)/sampl;
      Jbind_pm(:,irep)=Jbind_pm(:,irep)/sampl;
      Jind_pm(:,:,irep)=Jind_pm(:,:,irep)/sampl; %my g_pm
      tau_pm(1,irep)=tau_pm(1,irep)/sampl;
      if (strcmp(model_Phi,'equicorrelated')==1)
        rho_pm(1,irep)=rho_pm(1,irep)/sampl;
      end

   phatY_KL_irep=phat_running_total/sampl; % take average for KL calc
```

```
%%% calculate KL and L1
   %tic
   KL(1,irep)=distance_KL_irep_idX(phatY_KL_irep,log_pYT)
   L1(1,irep)=distance_L1(sigma_pm(:,:,irep), sigma_true)
   L1_SSY(1,irep)=distance_L1(SSY/nobs_adjusted, sigma_true) % uses SSY from ed's beta_vector

   %time_takenKL(irep)=toc,

end % for irep=1:nrep of the KL, L1 distances

%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%write out parameters of interest to file.

save DATAOUT/id17n40_eqsize_idXnrep20s20K
quit

%  save DATAOUT/hel_4cyclen17nobs20tauI_unif_idXnrep20s20K 1
%  save DATAOUT/hel_idn17nobs20tauI_size_idXnrep20s20K 2
%  save DATAOUT/hel_idn17nobs20tauSSY_unif_idXnrep20s20K 3
%  save DATAOUT/hel_idn17nobs20tauSSY_size_idXnrep20s20K
%  save DATAOUT/hel_idn17nobs20equicorr_unif_idXnrep20s20K
%  save DATAOUT/hel_idn17nobs20equicorr_size_idXnrep20s20K

%  save DATAOUT/hel_fulln17nobs20tauI_unif_idXnrep20s20K
%  save DATAOUT/hel_fulln17nobs20tauI_size_idXnrep20s20K
%  save DATAOUT/hel_fulln17nobs20tauSSY_unif_idXnrep20s20K
%  save DATAOUT/hel_fulln17nobs20tauSSY_size_idXnrep20s20K
%  save DATAOUT/hel_fulln17nobs20equicorr_unif_idXnrep20s20K
%  save DATAOUT/hel_fulln17nobs20equicorr_size_idXnrep20s20K

%  save DATAOUT/hel_tridiagn17nobs20tauI_unif_idXnrep20s20K
%  save DATAOUT/hel_tridiagn17nobs20tauI_size_idXnrep20s20K
%  save DATAOUT/hel_tridiagn17nobs20tauSSY_unif_idXnrep20s20K
%  save DATAOUT/hel_tridiagn17nobs20tauSSY_size_idXnrep20s20K
% save DATAOUT/hel_tridiagn17nobs20equicorr_unif_idXnrep20s20K
% save DATAOUT/hel_tridiagn17nobs20equicorr_size_idXnrep20s20K

%%write out parameters of interest to file.
%%save ../DATAOUT/var_cov_sel Jind_iter omega_iter sigma_iter ...
             %beta_vector_iter Jbind_iter Jind_pm...
             %omega_pm sigma_pm beta_vector_pm Jbind_pm
             %Tvec_iter Tvec_pm
             % L1 L1_SSY KL
```

8.1.24 miscellaneous subroutines.

argmax.m

The following 2 item list description is enumerated with respect to the 2 lines of MATLAB code that follows it. The code returns the index of the greatest element in the vector. In the case of ties, it returns the lowest index.

Although the code is trivial, it can be used to produce cleaner code in the calling programs. For example, in *check_chordal.m* the line

$$u = U(argmax(score))$$

which would otherwise require the two lines $[m, index] = max(score); u = U(index);$. Since the index of the greatest element is calculated in many of the calling programs, it is worth having this subroutine as a function.

1. use the inbuilt MATLAB function *max*.
2. index=the second output variable.

```
function index=argmax(v)
% input: a vector v

% output: the index of the greatest element.
%         Returns the first maximum in the case of ties.

   [m i]=max(v);
   index=i;
```

setdiag.m

The following 6 item list description is enumerated with respect to the 6 lines of MATLAB code (excluding comment and blank lines) that follows it. The code returns the original $p \times p$ matrix M with its diagonal entries replaced by the elements of v. v can be a scalar or a vector, or a $1 \times p$ or $p \times 1$ array.

1. find p, the dimension of M.
2. begin if test for v being scalar.
3. if v is a scalar, convert it to a $1 \times p$ vector for placing on the diagonal of M.
4. end if test to see if v is a scalar.

5. calculate J, the linear index string corresponding to the i, ith entries of M. For example, the $p, 1$th index is $J = n$, the $1, 2$th entry is $J = p + 1$, etc.

6. set M at all the indicies of J equal to v.

```
function M = setdiag(M, v)
% inputs: 1. M, a p x p matrix
%         2. v, a scalar, vector, or p x 1 or 1 x p array.

% output: 1. M with diagonal set to v.

   p = length(M);
   if length(v)==1
   v = repmat(v, 1, p);
         % repmat is an existing matlab function
   end
   J = 1:p+1:p^2;
      % J is the linear index corresponding to the i,ith entry of
      % M. For example, the (p,1)th index is J=n, the 1,2 entry is
      % J=p+1, etc
   M(J) = v;
```

intersect_zo.m

The following single item description is enumerated with respect to the single line of MATLAB code (excluding comment and blank lines) that follows it. The code finds the intersection of two subsets of variables assuming the subsets are each represented by a vector of zeros and ones. This works on the principle that the ith element of the elementwise multiplication $b = a1.*a2$ of two vectors $a1$ and $a2$ will be zero whenever either $a1(i) = 0$ or $a2(i) = 0$. Conversely, $b(i) = 1$ if and only if both $a1(i) = 1$ and $a2(i) = 1$. Thus b is a vector of zeros and ones, such that $b(i) = 1$ if and only if the associated variable is in both the subsets being intersected. For example, the intersection of $A = \{v_1, v_3, v_4\}$ with $B = \{v_2, v_3\}$ is given by
b=intersect_zo([1 0 1 1]' , [0 1 1 0]'), where *b=[0 0 1 0]'*.

1. take the elementwise multiplication of the two vectors.

```
function [b]=intersect_zo(a1, a2)
%  input:  a1, a2, a pair of [n,1] col vectors of zeros and ones only

%  output: b, an n x 1 col vector representation of the intersection
           between the subsets of variables represented by a1 and a2.

   b=a1 .* a2;
      %  .* is elementwise multiplication
```

```
      %  eg [0 0 1 0]'=intersect_zo([1 0 1 1]' , [0 1 1 0]')
      %  is equivalent to {v3}={v1, v3, v4} intersection {v2, v3}

% below is for error resistant version if required
% [n1,c1]=size(a1);
% [n2,c2]=size(a2);
%b=zeros(n1, c1);

%if (n1 ==n2) & (c1 == c2) & (c1 == 1)
   %b=a1 .* a2;
%end
```

union_zo.m

The following 6 item list description is enumerated with respect to the 6 lines of MATLAB code (excluding comment and blank lines) that follows it. The code finds the union of two subsets of variables assuming each subset is represented by a vector of zeros and ones. This works on the principle that the ith element of the elementwise addition $b = a1 + a2$ of two vectors $a1$ and $a2$ will be zero if and only if both $a1(i) = 0$ and $a2(i) = 0$. Conversely, $b(i) > 0$ if and only if at least one of $a1(i) = 1$ or $a2(i) = 1$. The routine replaces any $b(i) = 2$ with $b(i) = 1$. Thus b is a vector of zeros and ones, such that $b(i) = 1$ if and only if the associated variable is in one of the subsets in the union. For example, the union of $A = \{v_1, v_3, v_4\}$ with $B = \{v_2, v_3\}$ is given by
b=intersect_zo([1 0 1 1]' , [0 1 1 0]'), where *b=[1 1 1 1]'*.

1. take the sum of the two vectors.
2. begin for loop to replace any $b(j) > 1$ by $b(j) = 1$.
3. begin if test for $b(j) > 1$.
4. if $b(j) > 1$, set $b(j) = 1$.
5. end if $b(j) > 1$ test.
6. end for loop.

```
function [b]=union_zo(a1, a2)
%  input:  a1, a2, a pair of [n,1] col vectors of zeros and ones only

%  output: b, an n x 1 col vector representation of the union between
           the subsets of variables represented by a1 and a2.

%  b = elementwise addition, then make ones
```

```
%  eg [1 1 1 1]'=union_zo([1 0 1 1]' , [0 1 1 0]')
%  is equivalent to {v1, v2, v3, v4}={v1, v3, v4} union {v2, v3}

   b=v1 + v2;

   for j=1:length(b)
      if b(j)>1
         b(j)=1;
      end
   end

% below is for error resistant version if required
% [n1,c1]=size(v1);
% [n2,c2]=size(v2);
%b=zeros(n1, c1);

%if (n1 ==n2) & (c1 == c2) & (c1 == 1)
   %b=etc;
%end
```

set_diff_zo.m

The following 6 item list description is enumerated with respect to the 6 lines of MATLAB code (excluding comment and blank lines) that follows it. The code finds the set difference of two subsets of variables assuming each subset is represented by a vector of zeros and ones. This works on the principle that the ith element of the elementwise subtraction $b = a1 - a2$ of two vectors $a1$ and $a2$ will be one if and only if both $a1(i) = 1$ and $a2(i) = 0$. That is, if and only if the ith variable is an element of the subset represented by $a1$ but not of the subset represented by $a2$. If the ith variable is not an element of the set represented by $a1$, then $b(i) \leq 0$. The routine replaces any $b(i) < 0$ with $b(i) = 0$. Thus b is a vector of zeros and ones, such that $b(i) = 1$ if and only if the associated variable is in the set represented by $a1$, and not in the set represented by $a2$. For example, the setdifference between $A = \{v_1, v_3, v_4\}$ with $B = \{v_2, v_3\}$ is given by
b=set_diff_zo([1 0 1 1]' , [0 1 1 0]'), where *b=[1 0 0 1]'*.

1. take the difference $b = a1 - a2$. The zero/one representation ensures that the ith element of b is one if and only if $v_i \in a1$ and $v_i \notin a2$. Otherwise, the ith element of b is less than or equal to zero.

2. begin for loop to replace any $b(j) < 0$ by $b(j) = 0$.

3. begin if test for $b(j) < 0$.

4. if $b(j) < 0$, then $v_j \notin a1$, so the jth element of the zero one representation of the set difference should be zero.

5. end if $b(j) < 0$ test.

6. end for loop.

```
function [b]=setdiff_zo(a1, a2)
%  input:  a1, a2, an ORDERED pair of [n,1] col vectors
%          of zeros and ones only.
%          The ordering is such that the elements
%          represented by a2 are removed from the set
%          represented by a1.

%  output: b, an n x 1 col vector representation of the
%          set difference between the subsets of variables
%          represented by a1 and a2.

% b = elementwise addition, then make ones
% eg [1 0 0 1]'=setdiff_zo([1 0 1 1]' , [0 0 1 0]')
% eg [1 0 0 1]'=setdiff_zo([1 0 1 1]' , [0 1 1 0]')
% eg [0 0 0 0]'=setdiff_zo([1 0 0 1]' , [1 1 1 1]')

   b=a1 - a2;

   for j=1:length(b)
      if b(j)<0
         b(j)=a1(j);
       end
   end


% below is for error resistant version if required
% [n1,c1]=size(a1);
% [n2,c2]=size(a2);
%b=zeros(n1, c1);
%if (n1 ==n2) & (c1 == c2) & (c1 == 1)
   %b=etc;
%end
```

is_in_zo.m

The following 11 item list description is enumerated with respect to the 11 lines of MATLAB code (excluding comment and blank lines) that follows it. The code tests whether or not the integer a is a member of A. A can be input as a $p \times 1$ or $1 \times p$ array, or as a p dimensional vector. The function returns *element*= 1 if $a \in A$, and *element*= 0 otherwise. Note that A cannot be a cell array, and A is not a zero/one representation.

1. begin if test for cases.
2. if A is empty, then $a \notin A$.
3. otherwise test if $a < min(A)$, the smallest entry in A.
4. if $a < min(A)$, then $a \notin A$.
5. otherwise test if $a > max(A)$, the biggest entry in A.
6. if $a > max(A)$, then $a \notin A$.
7. if all the preceding tests fail, then test if $a \in A$.
8. initialise a *bits*, a $1 \times max(A)$ array, to zero.
9. for each member $\alpha \in A$, set $bits(\alpha) = 1$. *bits* is consequently a vector in which the αth entry equals 1 if $\alpha \in A$, and zero otherwise.
10. set $element = bits(a)$, the ath element of *bits*.
11. end if test for cases

```
function [element] = is_in(a, A)

% inputs: 1. A, a 1 x p or p x 1 array of integers.
%         2. a, the element to test whether a is a member of A.
% output: 1. element, a zero/one (yes/no) indicator of whether
%            or not a is a member of A.

% if isempty(A) | a < min(A) | a > max(A)  is slow

   if length(A)==0
      element = 0;
   elseif a < min(A)
      element = 0;
   elseif a > max(A)
      element = 0;
   else
     bits = zeros(1, max(A));
     bits(A) = 1;
     element = bits(a);
   end
```

mvt_gamma_ln.m

The below code calculates the natural logarithm of the multivariate gamma function $\Gamma(n, \alpha)$, $\alpha > (n-1)/2$, as defined on p.61 of Muirhead (1982). From Theorem 2.1.12, p.62 of Muirhead (1982) this can be expressed as the product of the ordinary gamma functions. The calculation in the code uses this product. The logarithmic version is necessary to avoid underflow/overflow problems

The code is self explanatory so the line by line description is omitted.

```
function [ln_mvt_gamma]=mvt_gamma_ln(n, alpha)
% returns the log of multivariate gamma(n, alpha) value.
% necessary for avoiding underflow/overflow problems
% alpha > (n-1)/2
% from Muirhead pp 61-62.
   sum_terms=0;
   for i=1:n
      term_i=gammaln(alpha-.5*(i-1));
      sum_terms=sum_terms+term_i;
   end

   ln_mvt_gamma=((n*(n-1))/4)*log(pi)+sum_terms;
```

ln_gam_pdf.m

This subroutine evaluates the density for $W = ln(X)$, where $X \sim \Gamma(a, b)$ has a gammadistribution with parameters a, b.

```
function [p_w, ln_p_w]=ln_gam_pdf(w, a, b)
p_w=exp(a*w)*exp(-b*exp(w))*b^a/gamma(a);
ln_p_w=a*w+-b*exp(w)+a*log(b)-gammaln(a);
% W=ln(X). Evaluates density for W=logX, where X is
% gamma(a,b)
```

8.1.25 Graphviz code and output

Download graphviz from http://www.graphviz.org/. Use MATLAB to input an adjacency graph. Use the MATLAB functions for undirected and directed graphs given in Figure 8.1.25 and Figure 8.1.25 respectively to create the required format files for graphviz. To create the .ps diagram, open an xterm in the same directory as your MATLAB output file, and type the relevant command from Figure 8.1.25 at the command prompt.

```
function undirected_graph(gg,outfile);
p = size(gg,1);
fid = fopen(outfile,'w');
```

```
  fprintf(fid,'graph G {\p');
  for i = 1:p
    for j = i:p
       if ( gg(i,j) == 1 )
          fprintf(fid,'%3.0f -- %3.0f [len=2];\p', i,j);
       end
    end
  end
  fprintf(fid,'}\p');
fclose(fid);
```

Undirected graph MATLAB interface for graphviz

```
function directed_graph(gg,outfile);
p = size(gg,1);
fid = fopen(outfile,'w');
  fprintf(fid,'digraph G {\p');
  for i = 1:p
    for j = i:p
       if ( gg(i,j) == 1 )
          fprintf(fid,'%3.0f -> %3.0f [len=2];\p', i,j);
       end
    end
  end
  fprintf(fid,'}\p');
fclose(fid);
```

Directed graph MATLAB interface for graphviz

```
 for an undirected graph:
     neato -Tps -o u_graph1.ps u_graph1

 for a directed graph:
     dot -Tps -o d_graph1.ps d_graph1
```

Xterminal commands for using *graphviz*.

8.2 Appendix B: Useful matrix theory

Theorem 8.2.1 *(Muirhead, 1982,Theorem A5.2, p.580) Let A be a $p \times p$ nonsingular matrix, and let $B = A^{-1}$. Partition A and B as*

$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, and B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$, *where A_{11} and A_{22} are nonsingular.*

Put $A_{11|2} = A_{11} - A_{12}(A_{22})^{-1}A_{21}$ and $A_{22|1} = A_{22} - A_{21}(A_{11})^{-1}A_{12}$.

Then $B_{11} = (A_{11|2})^{-1}$, $B_{22} = (A_{22|1})^{-1}$, $B_{12} = (A_{11})^{-1}A_{12}(A_{22|1})^{-1}$, and $B_{21} = (A_{22})^{-1}A_{21}(A_{11|2})^{-1}$.

8.3 Appendix C: Proofs of results

Lemma 8.3.1 *The space of decomposable graphs becomes increasingly sparse in the space of all graphs as $p = |V|$ increases.*

Proof. Let $V = \{v_1, \ldots, v_p\}$, and let A_p denote the number of decomposable graphs with vertex set V. It is easy to show that $A_4 = 61$ (see Section 7.3). Let B_p denote the number of simple undirected graphs on V. Clearly $|B_p| = 2^{\binom{p}{2}} = 2^{p(p-1)/2}$. Define a *random graph* $X_{p,1/2}$ as a simple undirected graph on p vertices where the probability of any edge is $1/2$ and the edges are independent. Let $\mathbb{P}$ be the product distribution over all edges $e \in E$ obtained as the product of the Bernoulli distribution on e with probability $1/2$. The probability $\mathbb{P}(g)$ that $X_{p,1/2} = g$ is clearly uniform, because if g is a graph on p vertices with m edges, then $\mathbb{P}(g) = \frac{1}{2}^m \frac{1}{2}^{\binom{p}{2}-m} = 2^{-\binom{p}{2}}$ which is independent of m. We need to show that $|A_p|/|B_p| \to 0$ as $p \to \infty$; i.e. $\mathbb{P}(g \text{ is chordal}) \to 0$ as $p \to \infty$. Partition the vertices as $C_1 = \{v_1, v_2, v_3, v_4\}, C_2 = \{v_5, \ldots v_8\}, \ldots, C_t = \{v_{4t-31}, \ldots, v_{4t}\}, C_{t+1} = \{v_{4t+1}, \ldots v_p\}$ where $p/4 - 1 \leq t \leq p/4$ is the largest integer less than or equal to $p/4$ and so $|C_{t+1}| \leq 3$. Let $g_{C_i}, i = 1, \ldots, t+1$ be the corresponding induced subgraphs, and note that $g_{C_{t+1}}$ must be chordal as it contains fewer than 4 vertices. Then $\mathbb{P}(g \text{ is chordal}) \leq \mathbb{P}(g_{C_1}, \ldots, g_{C_{t+1}} \text{ are all chordal}) \leq (61/64)^t \to 0$ as $p \to \infty$ because the induced subgraphs g_{C_i} of a chordal graph are all chordal, but an unchorded 4-cycle can be created by adding edges between distinct subgraphs g_{C_i} and g_{C_j}, where $j \neq i$. ■

Note that the bound is very conservative because there are many nonchordal graphs that have chordal subgraphs $g_{C_1}, \ldots, g_{C_{t+1}}$. For example, there many possible nonchordal cycles of length 4 or more that can be created by adding edges between distinct chordal subgraphs g_{C_i} and g_{C_j}, where $j \neq i$.

Proof of Theorem 4.8.1

Roverato (2000) shows that if $\Sigma \sim HIW(g, \delta, \Phi)$ and $\Omega = \Sigma^{-1}$ then

$$p(\Omega|g, \delta, \Phi) \propto |\Omega|^{(\delta-2)/2} \operatorname{etr}\left(-\frac{1}{2}\Omega\Phi\right) \tag{8.1}$$

The result then follows from (4.12) since

$$\begin{aligned} p(\Omega|y, g, \delta, \Phi) &\propto p(y|\Omega)p(\Omega|g, \delta, \Phi) \\ &\propto |\Omega|^{(n-1)/2} \operatorname{etr}\left(-\frac{1}{2}\Omega S_y\right) |\Omega|^{(\delta-2)/2} \operatorname{etr}\left(-\frac{1}{2}\Omega\Phi\right) \\ &= |\Omega|^{(n+\delta-3)/2} \operatorname{etr}\left(-\frac{1}{2}\Omega\left(S_y + \Phi\right)\right). \end{aligned}$$

Note that the conjugate prior result for Ω does not require the graph g to be decomposable.

Proof of Theorem 4.8.2

First

$$p(Y|\delta, \Phi, g) = \frac{p(Y|\Sigma, \delta, \Phi, g)p(\Sigma|\delta, \Phi, g)}{p(\Sigma|Y, \delta, \Phi, g)}.$$

The result then follows from (4.3), (4.4), (4.12) and Theorem 4.8.1.

Proof of Theorem 4.13.1

From Equation (5.23), Lemma 5.5 of Lauritzen (1996)

$$\Omega = \sum_{i=1}^{k} \left[\left(\Sigma_{C_iC_i}\right)^{-1}\right]^V - \sum_{i=2}^{k} \left[\left(\Sigma_{S_iS_i}\right)^{-1}\right]^V$$

and hence

$$E\left(\Omega|Y, \delta, \Phi, g\right) = \sum_{i=1}^{k} \left[E\left(\left(\Sigma_{C_iC_i}\right)^{-1}|Y, \delta, \Phi, g\right)\right]^V - \sum_{i=2}^{k} \left[E\left(\left(\Sigma_{S_iS_i}\right)^{-1}|Y, \delta, \Phi, g\right)\right]^V.$$

Now $\Sigma|Y, \delta, \Phi, g \sim \text{HIW}\left(\delta, \Phi^*, g^*\right)$, so from Dawid and Lauritzen (1993), if A is a complete set in g then $\left(\Sigma_{AA}\right)^{-1}|Y, \delta, \Phi, g \sim \text{Wishart}\left(\delta^* + |A| - 1, \Phi^*_{AA}\right)$. The result then follows from the properties of the Wishart distribution.

Proof of Lemma 7.4.2

1. For a nondecomposable graph to have 4 edges it must contain exactly one chordless 4-cycle and no other edges. There are $\binom{p}{4}$ possible choices for the 4 vertices, and for each choice of 4 vertices there are 3 different chordless 4-cycles.

2. For a graph to be nondecomposable with $\binom{p}{2} - 2$ edges it must contain exactly one 4 cycle and all other edges must be present. Then apply the proof of the above.

3. We can partition the nondecomposable graphs with 5 edges into 2 sets: (a) those with a chordless 5-cycle and no other edges, and (b) those with a chordless 4-cycle and an extra edge. For case (a) there are $\binom{p}{5}$ choices for the 5 vertices and for each choice there are $(5-1)!/2 = 12$ different chordless 5-cycles. For case (b) there are $\binom{p}{4} \times 3$ choices for the chordless 4-cycle, and for each choice of chordless 4-cycle there are $(\binom{p}{2} - 6)$ choices for the extra vertex pair constituting the edge.

8.4 Appendix D: FORTRAN code

```
MODULE graph_mod
! HELEN's VERSION runs perfectly last update 25/08/04
! TO DO: 1) replace n by nverts throughout,  VERY carefully
! 2) ensure all 'local' vars in routines are initialised
!  USE HARD COPY AND HIGHLIGHTER TO DO THIS
!
! Helen's versions indicated
! last helen new routine next_graph_gibbs compiles and works on A12k 500 reps k_max=10
! last helen new routine 17:32 17/8/04 check_edge_add_same_component: testing gibbs routine heaps
! and no stepping outside chordal space, so i think it's ok
! last helen new routine 14:30 7/8/04 check_edge_delete and worked ok

USE constants_mod
USE param_mod  ! NOTE: n is shared variable declared here
USE random_mod

IMPLICIT NONE

  INTERFACE argmax    ! NOTE: in calling routine can just call argmax, not one of specific
! versions below
     MODULE PROCEDURE r_argmax, i_argmax
  END INTERFACE

  INTERFACE setdiag
     MODULE PROCEDURE setdiag01, setdiag02
  END INTERFACE

CONTAINS
!!! ----------------------------------------------------------------------

FUNCTION i_argmax(v)
! input: 1. v, vector of integers
! output: returns only one number=arg of first answer if repeated inclusions

implicit none
   integer, intent(in), dimension(:) :: v
   integer, intent(out) :: i_argmax
```

```
   integer :: v1(1)
   v1=maxloc(v, MASK=.TRUE.)
         !the MASK argument set to TRUE is default:= "consider all elts"
         !i.e. the "mask" covers some parts of the vector v
   i_argmax = v1(1)
END FUNCTION i_argmax
!!! ------------------------------------------------------------------------
FUNCTION r_argmax(v)
implicit none
real(wp), intent(in), dimension(:) :: v
integer :: r_argmax
integer :: v1(1)
v1=maxloc(v,MASK=.TRUE.)
r_argmax = v1(1)
END FUNCTION r_argmax

!!! ------------------------------------------------------------------------

FUNCTION is_in(a,B)

! input: 1. B is zero one representation of NON-ORDERED set of
!           non-zero integers. eg B=[1 0 1 1 0] ~ [3, 1, 4]
!        2. a is NON-ZERO element to find
! output: 0/1 for FASLSE/TRUE eg is_in(3,B)=1, is_in(5,B)=0

implicit none
integer, intent(in), dimension(:) :: B
integer, allocatable, dimension(:) :: bits
integer :: a, element, is_in
integer :: minB, maxB, i

element = 0
if ( B(a) == 1 ) then
    element = 1
endif

is_in = element
END FUNCTION is_in

!minB = minval(B,DIM=1,MASK=.TRUE.)
!maxB = maxval(B,DIM=1,MASK=.TRUE.)
!if (size(B,1)==0 ) then
!  element = 0;
!elseif (a < minB ) then
!  element = 0;
!elseif (a > maxB ) then
!  element = 0;
!else
!  allocate(bits(maxB))
!  bits = 0.0
```

```
!  bits(B) = 1;
!  element = bits(a);
!endif

!!! ------------------------------------------------------------------------
SUBROUTINE next_edge_candidate(g,edge_candidate)
!
! input: 1. g is [n,n] array where n is SHARED varibale defined in param_mod
! output: 1. edge_candidate is 1-di vector, 2 elts = edge vertices

integer, intent(in), dimension(:,:) :: g
integer, intent(out), dimension(2):: edge_candidate

integer, dimension(n) :: iota
integer :: i,j

iota=0
call randperm(iota,n)
i=iota(1); j = iota(2);
edge_candidate= iota(1:2)


END SUBROUTINE next_edge_candidate
!!! ------------------------------------------------------------------------
SUBROUTINE reachability_graph(g,reach_graph)

! input: 1. g is [n,n] array where n is SHARED varibale defined in param_mod
!
! output: 1. reach_graph is [n,n] arrary of all nodes reachable

integer, intent(in), dimension(:,:) :: g
integer, intent(out), dimension(:,:) :: reach_graph

! local variables
integer, dimension(n,n) :: A,B

integer :: i

A(n,n)=0
   B(n,n)=0

   A = g;
   B = 0
   do i=1,n-1
     B = B + A;
     A = matmul(A,g)
   enddo
   reach_graph = 0
   where (B /= 0) reach_graph = 1
```

```
END SUBROUTINE reachability_graph
!!! ------------------------------------------------------------------------

!  SETDIAG NOTES: Fortran automatically calls the right version of the below
!  two SETDIAG subroutines, via the "interface" declaration at the very beginning of
!  the modules.
!  NOTE: no need to say 01/02 in call
!!! ------------------------------------------------------------------------
SUBROUTINE setdiag01(M,v)
! DON'T USE THIS VERSION
! this version overwrites M with M+eye(n)
integer, intent(inout), dimension(:,:) :: M
integer, intent(in), dimension(:) :: v

integer :: i,j

if (size(v,1) == 1 ) then
   do i = 1 , n
     M(i,i) = v(1)
   enddo
else
   do i = 1 , n
      M(i,i) = v(i)
   enddo
endif

END SUBROUTINE setdiag01

!!! ------------------------------------------------------------------------
SUBROUTINE setdiag02(M,v,Mdiag)
!
! this version does NOT overwrite M
!
integer, intent(in),dimension(:,:) :: M
integer, intent(in), dimension(:) :: v
integer, intent(out), dimension(:,:) :: Mdiag
integer :: i,j

Mdiag=M

if (size(v,1) == 1 ) then
   do i = 1 , n
     Mdiag(i,i) = v(1)
   enddo
else
   do i = 1 , n
      Mdiag(i,i) = v(i)
   enddo
endif
```

```
END SUBROUTINE setdiag02

!!! -----------------------------------------------------------------------
SUBROUTINE setdiff(A,B,C)

! input: 1. A(p), B(p) are each a NON-ORDERED zero one representations of sets of
!           non-zero integers, but the ORDER of the call matters a lot
!           setdiff(A,B,C)= A\B. eg B=[1 0 1 1 0] ~ [3, 1, 4]
! output:2. C(p) is NON-ORDERED zero one representation of setdiff(A,B)
!            eg setdiff(B, [1 1 0 0 0])=[0 0 1 1 0] ~ [3,4]

integer, intent(in), dimension(:) :: A,B
integer, intent(out), dimension(:) :: C

C = A - (A*B)

END SUBROUTINE setdiff
!!! -----------------------------------------------------------------------
SUBROUTINE setunion(A,B,C)

! input: 1. A(p), B(p) NON-ORDERED zero one representations of sets of
!           non-zero integers. eg B=[1 0 1 1 0] ~ {3, 1, 4}
! output:2. C(p) is NON-ORDERED zero one representation of union(A,B)
!            eg setunion(B, [1 1 0 0 0])=[1 1 1 1 0] ~ {3,2, 1, 4}

integer, intent(in), dimension(:) :: A,B
integer, intent(out), dimension(:) :: C

C = A + B - (A*B)  ! fortran * is elementwise

END SUBROUTINE setunion
!!! -----------------------------------------------------------------------
SUBROUTINE setintersect(A,B,C)
! input: 1. A(p), B(p) NON-ORDERED zero one representations of sets of
!           non-zero integers. eg B=[1 0 1 1 0] ~ [3, 1, 4]
! output:2. C(p) is NON-ORDERED zero one representation of intersection(A,B)
!            eg setintersect(B, [1 1 0 0 0])=[1 0 0 0 0] ~ {1}

integer, intent(in), dimension(:) :: A,B
integer, intent(out), dimension(:) :: C

C = A*B  ! fortran * is elementwise

END SUBROUTINE setintersect
!!! -----------------------------------------------------------------------
SUBROUTINE indexes_of_nodes(vectorA, indicies)

! created by Helen 6/8/04 17:55 and works fine

! SHOULD I MAKE max_index assumed to be n thoughout?
```

```
! aim:  mimic matlab's find(vectorA==1) by using vectorA as the mask in pack
! input:  1. vectorA is zero one representation of NON-ORDERED set of
!           non-zero integers. eg vectorA=[1 0 1 1 0]= a clique or separator
! output: 1. indicies is sum(vectorA) length vector of indicies where vectorA==1
!         eg find_indicies_equal_one(vectorA, indicies)=[3, 1, 4]

!! NOTE: prior to calling this must initialise
!! indicies as an allocatable vector of length=sum(vectorA)

integer, intent(in), dimension(:) :: vectorA
integer, intent(out),dimension(:) :: indicies

integer, allocatable, dimension(:) :: all_indicies

integer :: i, max_index, num_ones

max_index=size(vectorA, 1)
num_ones=sum(vectorA)

allocate(all_indicies(max_index) )
all_indicies = 0
all_indicies = (/(i,i=1,max_index)/)

indicies=0
indicies=pack(all_indicies, vectorA /= 0 )
   ! could alternatively write pack(all_indicies, vectorA /=0, indicies)

deallocate(all_indicies)

END  SUBROUTINE indexes_of_nodes
! ------------------------------------------------------------------------

SUBROUTINE parents_node(g,a,ps)
! input: dimension(n,n) adjacency matrix g of graph and node a whose parents are
!       required.
!outputs 1: dimension(n) vector ps of parent nodes
!
!  NOTE: This only works if nodes in graph are ordered as per adj_mat.
!        i.e. if mcs order is 1, 7, 3, 4, 2, ... then it will return
!           [1 0 1 0 0 0 0]'=parents(adj_mat, 4) (=[1,3])
!               which is WRONG (should be 1 7 3)

integer, intent(in), dimension(:,:) :: g
integer, intent(in) :: a
integer, intent(out), dimension(:) :: ps

ps = g(a,:)
ps(a:n) = 0
```

```
END SUBROUTINE parents_node
! ps=zeros(n,1);
! nbs=neighbours_node_zo(g, a);
          ! works fine for no_cell version, as neighbours_node.m
          ! only uses vectors (no cell arrays)
! ps=nbs;
! ps(a:n,1)=0;
!!! ------------------------------------------------------------------------
SUBROUTINE neighbours_node(g,r,nbs)
! input: dimension(n,n) adjacency matrix g of graph and node r whose parents are
!       required.
!outputs 1: dimension(n) vector of connected nodes (=neighbours)
!
!  NOTE: This works for any mcs ordering of nodes as nbs are parents or descendants

integer, intent(in), dimension(:,:) :: g
integer, intent(in) :: r
integer, intent(out), dimension(:) :: nbs
   nbs = g(:,r)
END SUBROUTINE neighbours_node
!!! ------------------------------------------------------------------------
SUBROUTINE check_chordal(G,chordal, order)

integer, intent(in), dimension(:,:) :: G
integer, intent(out),dimension(:) :: order
integer, intent(out) :: chordal

integer, dimension(n) :: numbered, iota, ones, U
integer, dimension(n) :: ulist, uint
integer, dimension(n) :: nns, ordertmp
integer, dimension(n,n) :: Gdiag

integer, allocatable, dimension(:) :: score, Upack


integer, dimension(1) :: jj  ! needed as maxloc returns 1-di array, NOT a scalar
integer :: i,j,sumU,usmall

Gdiag=0
iota=0; U=0
ulist=0; uint=0; nns=0;

numbered = 0
ordertmp = 0
ones = 1
iota = (/(i,i=1,n)/)
Gdiag=0

call setdiag(G,(/1/), Gdiag);  ! if use the setdiag01 which writes over G, must re-set G
                               ! change G back into same as beginning with setdiag(G,(/0/))
```

```
order = 0
chordal = 1;
numbered(1) = 1;
order(1) = 1;
do i=2,n
    call setdiff(ones, numbered,U);
    sumU = sum(U,1)
    allocate(score(sumU))
    allocate(Upack(sumU))
    score = 0
    Upack = pack(iota, U /= 0)
    do j=1,sumU
      usmall = Upack(j);
      call neighbours_node(Gdiag,usmall,ulist)
      call setintersect(ulist,numbered,uint)
      score(j) = sum(uint,1)
    enddo
    deallocate(Upack)

    jj=maxloc(score)
    sumU = sum(U,1)
    allocate(Upack(sumU))
    Upack = pack(iota, U /= 0)
    usmall = Upack(jj(1));
    numbered(usmall) = 1
    order(i) = usmall;
    deallocate(Upack)

    ordertmp = 0
    where (order /= 0 ) ordertmp(order) = 1
    ordertmp(i:n) = 0

    call neighbours_node(Gdiag,usmall,ulist)
    call setintersect(ulist,ordertmp,nns)

    sumU = sum(nns,1)
    allocate(Upack(sumU))
    Upack = pack(iota, nns /= 0)

    if ( sum(sum(Gdiag(Upack,Upack),1),1) /= sumU**2  )  then
      chordal = 0
      print*, "chordal = " , chordal
      return ;
    endif
    deallocate(Upack,score)
enddo

!call setdiag(G,(/0/)); ! alternative if use the "write over G version", so that
                        ! G goes back to the same as passed in
```

```
! print*, "chordal = " , chordal
! write(*,'(A,9(I3,2x))') "order", order

END SUBROUTINE check_chordal
!!! ------------------------------------------------------------------------
SUBROUTINE chordal_to_ripcliques(g, order, cliques)
! created by Helen off _zo.m version
! last update Fortran: 6/8/04 14:18 AND IT WORKS!!!!!
!function [cliques, ladder]=chordal_to_ripcliques_no_mcs_zo(g, order)


! input:  1. a graph with nodes and corresponding adj_mat=g
!         2. mcs ordering, must be output of check_chordal
! output: 1. [n,n] cliques of g, st cliques(:,i) is the ith RIP ordered clique
!            RIP is as per perfect mcs ordering of nodes. if
!            order(j) is the ith ladder node, the ith column is the associated clique.
!            (has zero cols padding to dimension [n,n] at the end
!         2. a vector of the ladder nodes, ordered as per mcs ordering
!            calculated in the subroutine, with ladder(j)=order(j).
!

integer, intent(in), dimension(:,:):: g
integer, intent(in), dimension(:) ::  order

integer, intent(out), dimension(:,:):: cliques

integer, dimension(n)   :: pre_v, ns_v, pre_ns_v
   integer, dimension(n)   :: num_pre_ns, ladder
   integer, dimension(n)   :: v_col_i, cliques_index_non_zero_ladder
integer, dimension(n, n):: pre_ns

integer :: v,i,j, index_non_zero_ladder

ladder=0
pre_ns=0
num_pre_ns=0
!    0/1 matrix st pre_ns(:,i)=predecessor neighbs of node i in g
!                              = parents of node i in g wrt mcs order of g,
!                           NOT wrt index ordering of adjacency matrix

do i=2,n
      v=order(i);  ! select the ith node in the ordering, i=2, n

      pre_v=0      ! initialise i dependent vectors to 0 for each i
      ns_v=0       ! do NOT deallocate/allocate within loop as same size each i
      pre_ns_v=0   ! these are i dependent as v=order(i), so v is order(i)^th node
                   ! with respect to adjacency matrix and cliques matrix ordering

      pre_v(order(1:i-1))=1
! in fortran, CAN have vector( [3, 1, 7, 9])=value, and will do right thing
```

```
! = 0/1 vector representing predecessors of v, the ith node in mcs ordering.
            ! NOT same as parents as predecessors in order are not necess. neighbours in g

call neighbours_node(g, v, ns_v)
               !ns_v= 0/1 representation of set of neighbours of each v=order(i)

      call setintersect(ns_v, pre_v, pre_ns_v);

      num_pre_ns(i)=sum((pre_ns_v), 1); != number of pre_neighbours of v=ith mcs node
      pre_ns(:, v)=pre_ns_v;
!= 0/1 matrix representation of set of neighbours of each v=order(i)
! do NOT reset these to zero on each i, as they are vals for all i


        ! must NOT deallocate pre_ns(n,n) and num_pre_ns(n)
        ! note these are allocated before do i=1, n endo

          !find the sets of those neighbours which precede
          !v=order(i) with respect to order. Store answer for cliques.
          ! so if order=[1 3 7 5 2 4 6], pre_ns(:,order(4)=5)=[0 0 1 0 0 0 1]'
          ! pre_ns =
              !      0     0     1     0     0     0     1
              !      0     0     0     1     0     0     0
              !      0     0     0     0     1     0     1
              !      0     0     0     0     0     0     0
              !      0     0     0     0     0     0     0
              !      0     0     0     0     0     0     0
              !      0     1     0     0     1     1     0
          ! num_pre_ns ordered as per order (i)  = 0 1 2 2 1 1 1
          ! corresponding to v= 1 3 7 5 2 4 6

          ! eg num of pre nbs of 3 for ladder test=sum(:,3), etc.
          ! note that the ith column
          !corresponds to the pre-nbs of the ith node in order;
          !i.e v=order(i), and NOT v=i. Simly for cardinality
          !Need to keep variables ordered as per the
          !mcs ordering, the vector called order.
enddo

   !! NOTE: for ladder(i)~=0, ladder(i)=order(i)

do i=1, n
ladder(i)=0
    if ( i==n .or. (num_pre_ns(i) >= num_pre_ns(i+1)) ) then
! num_pre_ns(i)=sum(pre_ns (:, order(i)))
            !              = number of predecessor nbs of v=order(i)
            !
            ! if i=n or cardinality of pre_ns decreasing with
            ! v=order(i), then the node v=order(i) is a ladder node.
            ! eg, i=4, order as above has ladder [0 0 7 5 2 4 6]
```

```
            ! as == or decrease occurs num_pre_ns(i), i=3 4 5 6 7
            ! and order(3)=7,order(4)=5,order(5)=2,order(6)=4,order(7)=6.
       ladder(i)=order(i) !make this v the next ladder node
    endif
enddo

!!!!!!!!!!!!!!!!!!!!!!
!!! NOTE: i output non-empty columns first, so clique ordering follows
!! matrix cliques column ordering.
!! if only 3 cliques, then cliques(:, 3:n)=zero(n,1);
!! cliques(:,i)=union of the ith non-zero ladder node and its predecessors)
!! eg cliques(:,3)=[0 1 0 0 0 0 0]+[0 0 0 0 0 0 1] = {2} U {7}
!! NOTE: for ladder(i)~=0, ladder(i)=order(i)


! num_cliques=size(find(ladder), 2);
! could find num_cliques via empty_cliques=like farids pack
! see   check_chordal routine

index_non_zero_ladder=0

! initialise i dependent vars
   v_col_i=0
   cliques_index_non_zero_ladder=0

do i=1,n
    if (ladder(i) /= 0) then
         v_col_i=0   ! initialise new column vector rep. of v=ladder(i)
         v_col_i(ladder(i))=1; ! ladder(i)=order(i)
         index_non_zero_ladder=index_non_zero_ladder+1;

         cliques_index_non_zero_ladder=0
!initialise clique with index of next NON_ZERO ladder node
         call setunion(v_col_i,pre_ns(:,ladder(i)), cliques_index_non_zero_ladder)
         cliques(:, index_non_zero_ladder)=cliques_index_non_zero_ladder

    endif
enddo

! Theorem: cliques(index_non_zero_ladder, 1:size_cliques) are the cliques of g
! and clique ordering satisfies the RIP.

END SUBROUTINE chordal_to_ripcliques
!!! -----------------------------------------------------------------------
SUBROUTINE cliques_number_and_sizes(cliques, cliques_sizes, num_cliques)
! created by Helen 070804 14:50 and runs perfectly

! aim: return number of non-empty cliques, and sizes of each
! input: [n,n] matrix st
!        clique_i=cliques(:,i), cliques(v,i)=0/1 indicator of node v in clique i
```

```
! output: 1. clique_sizes(n) st clique_sizes(i)= no. nodes in clique i
!         2. num_cliques=number of non-empty cliques

integer, intent(in), dimension(:,:) :: cliques
integer, intent(out), dimension(:)  :: cliques_sizes
integer, intent(out)    :: num_cliques

integer, allocatable, dimension(:) :: cliques_indicator

cliques_sizes=0  ! note this can have zeros for final cols if they are empty
cliques_sizes=sum(cliques, 1) ! this is a 1xn vector of column totals

allocate(cliques_indicator(size(cliques_sizes)))

cliques_indicator=0
where (cliques_sizes > 0) cliques_indicator=1
num_cliques=sum(cliques_indicator,1)

deallocate(cliques_indicator)
END SUBROUTINE cliques_number_and_sizes
!!! ------------------------------------------------------------------------
SUBROUTINE non_empty_columns(mat_empty_cols, nnon_empty, mat_non_empty)

! aim: takes [n,n] matrix possibly containing all zero columns, and returns
! nnon_empty=number of non-empty columns (can be zero if all have element)
! and mat_non_empty=[n,nnon_empty] matrix = mat_empty_cols without the all
! zero columns.
! NOTE: this is NOT same as fortran "pack" which just makes it a 1-di array

integer, intent(in), dimension(:,:) :: mat_empty_cols
integer, intent(out), dimension(:,:):: mat_non_empty
integer, intent(out)                :: nnon_empty
integer  :: i, p, count, first_empty

integer, dimension(n) :: column_i

      i=0
p=size(mat_empty_cols,2)
      count=0
      mat_non_empty=0
      nnon_empty=0
      first_empty=0

      column_i=0

do i=1, p
         column_i=mat_empty_cols(:,i)
         if (sum(column_i) == 0) then
            count=count+1
            if (count == 1) then
```

```
              first_empty=i
               !!!!exit
            endif
         endif
      enddo

      nnon_empty=p-count
      mat_non_empty=mat_empty_cols(:, 1:first_empty-1)


END SUBROUTINE non_empty_columns
!!! ------------------------------------------------------------------------

SUBROUTINE num_non_empty_columns(mat_empty_cols, nnon_empty)

! aim: takes [n,n] matrix possibly containing all zero columns, and returns
! nnon_empty = index of last non_empty (can be zero if all have element)
! NOTE: this is NOT same as fortran "pack" which just makes it a 1-di array

integer, intent(in), dimension(:,:) :: mat_empty_cols
integer, intent(out)                :: nnon_empty
integer  :: i, p, count, first_empty

integer, dimension(n) :: column_i

      count=0
      nnon_empty=0
      first_empty=0

      column_i=0

do i=1, size(mat_empty_cols,2)
         column_i=mat_empty_cols(:,i)
         if (sum(column_i) == 0) then
            count=count+1
            if (count == 1) then
              first_empty=i
              exit
            endif
         endif
      enddo

      nnon_empty=p-count

END SUBROUTINE num_non_empty_columns
!!! ------------------------------------------------------------------------

SUBROUTINE ripcliques_to_jtree(cliques,jtree)
! this is HELEN's version
```

```
! input:   1. cliques=[n,n] matrix zero/one output of chordal_to_ripcliques_zo
! output:  2. jtree=[n,n] matrix array of associated junction tree.
!
integer, intent(in), dimension(:,:) :: cliques
integer, intent(out), dimension(:,:) :: jtree

integer, dimension(n) ::  score, clique_i, sep_ik
integer :: i, j, k

clique_i=0; sep_ik=0
score=0
jtree=0

do i =2, n
    clique_i=cliques(:, i)
      if ( sum(clique_i) > 0 ) then
      do k=1, i-1
       call setintersect(cliques(:,i), cliques(:,k), sep_ik)
       score(k)=sum(sep_ik)
           ! must ONLY intersect with predecessor cliques in the RIP ordering
   enddo

      if ( maxval(score) .ne. 0 ) then
      ! only add the edge if clique i IS connected to one of its
      ! predecessors. if score is all zeros, then clique has no intersection
      ! with any of its predecessors. Since the cliques are in RIP, it must
      ! follow that we are no longer in the same connected component of
      ! the graph.
      ! if clique i has no intersection with any of the preceding cliques,
      ! then the graph is disconnected, so the adjacency matrix will have
      ! a zero row/column for this i, and we have a forest, not a j_tree.

      ! add edge_ij between clique_i and the clique preceding it in the order with
      ! which has most intersection with clique_i

      j=0
      j=argmax(score)
      jtree(i,j)=1
      jtree(j,i)=1
    endif
endif
enddo

END SUBROUTINE ripcliques_to_jtree
!!! ------------------------------------------------------------------------
SUBROUTINE separator_sizes(cliques,jtree,sepsize)
! HELEN'S VERSION
! input:   1. cliques=[n,n] matrix zero/one output of chordal_to_ripcliques_zo
!          2. jtree=[n,n] the associated junction tree.
! output:  1. a matrix array of the size of the separator sets, where
```

```
!              sepsize(i,j) is the number of elements in the separator set between
!              adjacent cliques i and j with respect to jtree.

! NOTE: if you want a 1xnum_seps array of the separators, use
! seps_residuals_histories.m

integer, intent(in), dimension(:,:) :: jtree
integer, intent(in), dimension(:,:) :: cliques
integer, intent(out), dimension(:,:) :: sepsize

integer, dimension(n) :: sep_kj
integer :: k, j


sep_kj=0
sepsize=0

do k=1, n-1
  !!if (jtree(:,i) .eq. 0) then  ! NO this will be satisfied for disconnected jtrees
       !!exit                   ! for disconnected graphs
  !!end
  do j=k, n
   if (jtree(k,j) .eq. 1) then
          sep_kj=0
          call setintersect(cliques(:, k), cliques(:, j), sep_kj)
          sepsize(k,j)=sum(sep_kj);
    sepsize(j,k)=sum(sep_kj);
      endif
   enddo
enddo


END SUBROUTINE separator_sizes

!!! -------------------------------------------------------------------------
SUBROUTINE next_graph_candidate(g,jtree,sepsize,cliques,reach_graph, &
                           g_proposal, a,b, new_clique, CASE_add, CASE_delete);

integer, dimension(:,:) :: g,jtree,cliques,sepsize,reach_graph
integer, dimension(:,:) :: g_proposal
integer, dimension(:) :: new_clique
integer ::  a,b, CASE_add, CASE_delete
integer, dimension(n) :: new_clique_potential_add, new_clique_potential_delete
integer, dimension(2) :: v
integer :: CASE1,CASE2
integer :: i,j

new_clique_potential_add=0
new_clique_potential_delete=0
new_clique = 0
```

```
i=0; j=0; v=0; g_proposal=g

CASE_add=0; CASE_delete=0;
a=0; b=0;
new_clique_potential_add=0; new_clique_potential_delete=0;
CASE1=0; CASE2=0;

!print*, ALL (g_proposal == g)

do while ( ALL (g_proposal == g) )
  call next_edge_candidate(g,v)
  i=v(1); j=v(2);

  if (g(i,j)==1 ) then
     call check_edge_delete(i,j, cliques, CASE_delete, new_clique_potential_delete)
  endif

  if ( g(i,j)==0 .and.  reach_graph(i,j)==0 ) then
      CASE1=1;
      CASE_add=1; a=i; b=j;
      new_clique(a)=1
      new_clique(b)=1
      g_proposal(i,j)=1; g_proposal(j,i)=1;
  elseif ( g(i,j)==0 .and. reach_graph(i,j) ==1 ) then
      call check_edge_add_same_component(i,j,jtree,sepsize,cliques,CASE_add, new_clique_potential_add)
  endif

  if ( g(i,j)==1 .and.  CASE_delete==1) then
      g_proposal(i,j)=0; g_proposal(j,i)=0;
      CASE_delete=1; a=i; b=j; new_clique=new_clique_potential_delete;

  elseif (g(i,j)==0 .and. reach_graph(i,j) ==1) then
      if ( CASE_add==1 ) then
        a=i; b=j; new_clique=new_clique_potential_add;
        g_proposal(i,j)=1; g_proposal(j,i)=1;  CASE2=1;
      endif
  endif

enddo

END SUBROUTINE next_graph_candidate
!!! ------------------------------------------------------------------------
SUBROUTINE check_edge_delete(v1,v2, cliques, delete_ok, clique_with_edge)

! created by Helen: 070804 16:15

!inputs: 1. v1, v2 the edge vertices (must exist)
!        2. the cliques=[n,n] matrix output of chordal_to_ripcliques in RIP
!           according to the mcs order
```

```
! outputs: 1. delete_ok=0/1, indicator for whether or not deletion ok
!          2. clique_with_edge(n) indicator col_vec of the clique
!             which contains the edge v1, v2

!NOTE: if edge doesn't exist, you can "delete it and remain chordal", so
!output is 1. however, don't want to count this as a move.

integer, intent(in), dimension(:,:) :: cliques
integer, intent(in) :: v1,v2
integer, intent(out), dimension(:)  :: clique_with_edge
integer, intent(out) :: delete_ok
integer :: k, counter, num_cliques, size_clique_k

! local variables
integer, dimension(n)  :: cliques_sizes, clique_k, edge_zo,edge_int_clique

k=0
delete_ok=1
! initialise to ok, and make zero if "not_ok condition" NOT satisfied
! in below do loop
clique_with_edge=0
counter=0
! initialise number of cliques containing the edge i,j to 0
! but note that edge MUST be inside a clique for valid use of
      ! subroutine (every edge is in one clique)

!!! find the sizes of each clique, and the number of non-emtpy cliques

cliques_sizes=0
call cliques_number_and_sizes(cliques, cliques_sizes, num_cliques)

!!! create col vector representation of edge [v1,v2],
!!! and initialise intersection vector

   edge_int_clique=0
edge_zo=0
edge_zo(v1)=1
edge_zo(v2)=1

do k=1,num_cliques
clique_k=0
    clique_k=cliques(:,k)
size_clique_k=0
    size_clique_k=cliques_sizes(k)
       call setintersect(edge_zo, clique_k, edge_int_clique )

if (sum(edge_int_clique)==2) then
       ! since cliques are by definition complete, if a clique contains
       ! vertices i and j, then it contains the edge i,j.
```

```
       clique_with_edge=clique_k; ! if it's ok to delete the edge, this will be the clique it's in.
       counter=counter +1

if (counter > 1) then  ! could alternatively have if counter == 2, since need 1 for legal
          delete_ok =0
clique_with_edge=0
          exit  ! exit does what i want: exits both nested internal if, AND the do loop
       endif
    endif
enddo

! THEORY: can ONLY delete edges that are in ONE SINGLE clique (else clique is in separator)
! draw picture and can see why

END SUBROUTINE check_edge_delete

!!! ------------------------------------------------------------------------
SUBROUTINE find_clique_containing(a,cliques,index_a)
! purpose: finds from nxn array of cliques, the index of 1st clique containing a

!NOTE: a MUST be inside one of the cliques or the routine never breaks.

! input:    1. the vertex a
!           2. cliques(n,n) zero/one matrix of NON-EMPTY cliques
! output:  1. the index of the first clique (in the order of the matrix array)
! that contains a

integer, intent(in) :: a
integer, intent(out) :: index_a
integer, intent(in), dimension(:,:) :: cliques

integer :: i,j,found

i=1; j=1; found=0;
index_a=1;
do while (found==0)
    if (cliques(a,i)==1 ) then
       index_a=i; found=1; return;
    endif
    if (found == 0) then
      i=i+1
    endif
enddo
END SUBROUTINE find_clique_containing

!!! ------------------------------------------------------------------------
SUBROUTINE check_edge_add_same_component(a,b, jtree, sepsize, cliques, edge_ok, new_clique)
! created 13/08/04 Helen Armstrong off _zo.m equivalent
! latest update 19/08/04 12:45 and is running for 30K iterations of the
! next_graph_gibbs call. so far, no problems on 18 nodes g4discon2.dat
```

```
! input:   1. vertices a, b of the edge to be added. g(a,b) MUST be zero. a, b MUST be
!   in SAME connected component of g.
!          2. [n,n] associated junction tree from ripcliques_to_jtree
!          3. matrix array of the size of the separator sets, where
!              sepsize(i,j) is the number of elements in the separator set between
!              adjacent cliques i and j with respect to jtree.
!          4. cliques=[n,n] matrix output of chordal_to_ripcliques
! output:  1. edge_ok=0(no)/1(yes)
!          2. [n,1] new_clique, the new clique

! NOTE: only works for mcs output cliques,  and in same connected component.
! NOTE: ensure subroutine ONLY called by the main program if the nodes are in
! same connected component.

integer, intent(in) :: a,b
integer, intent(in), dimension(:,:) :: cliques, jtree, sepsize
integer, intent(out), dimension(:) :: new_clique
integer, intent(out) :: edge_ok

! print*, 'line 64'

! local variables
integer :: a2, b2, s, bottom, max_iterations, num_parents
integer :: tree_top, CASE_sat_on_same_branch,CASE_same_branch
integer :: CASE_sats_on_a2_branch, CASE_sats_on_b2_branch
   integer :: index_a, index_b, index_a2, index_b2, next_index, index
integer :: next_parent_b2, next_parent_a2
   integer :: next_ancestor_a2, next_ancestor_b2, num_int
integer :: count, index_row, index_col, num_seps_branch_a2, num_seps_branch_b2
integer :: fork, fork_index
integer :: locate_a2
integer :: quit, i, j, dummy
integer, dimension(n) :: clique_next_index
integer, dimension(n) :: ab_edge_vec, parent, clique_a2_int_clique_b2_zo
integer, dimension(n) :: ancestors_a2_zo, ancestors_b2_zo
integer, dimension(n) :: anc_a2_int_anc_b2_zo

integer, allocatable, dimension(:) :: ancestors_a2, ancestors_b2, anc_a2_int_anc_b2
! need ancestors and anc_a2_int_anc_b2 as a numeric i.e. NOT zero one rep

! initialise variables
new_clique=0
edge_ok=0
tree_top=0
fork=0
CASE_sat_on_same_branch=0
CASE_same_branch=0
CASE_sats_on_a2_branch=0
CASE_sats_on_b2_branch=0
```

```
   locate_a2=0
!initialise top of component to "not found". DON'T use 1, as  the
!top might be >1 in a disconnected graph.

ab_edge_vec=0
ab_edge_vec(a)=1
ab_edge_vec(b)=1

quit = 0
i=0
j=0

!! Rename nodes so that a2<b2, and a2 in clique(index_a2), b2 in clique(index_b2),
!! Note that jtree indicis correspond to nodes of cliques in RIP, i.e. order of
!! node=clique is same as column index of clique.

call find_clique_containing(b, cliques, index_b)
   call find_clique_containing(a, cliques, index_a);
!print*, 'a, b=', a, b, 'index_a, index_b=', index_a, index_b

  index_b2=max(index_a,index_b)

!print*, 'max(index_a,index_b)', max(index_a,index_b)
    !index_b2 >1, since max[ ]>1 (Can't have both a and b in first clique, as
    ! there is no edge between them. Also, |index_a -index_b|>=1.
    ! hence below if..else ensures correct ordering of index_a2, index_b2
    ! i.e. know clique(index_a2, :) precedes index_b2

! Re_name nodes so b is in clique(index_b2, :), a is in clique(index_a2, :);
if (index_b2==index_b) then
 b2=b
 a2=a
else
b2=a
a2=b
end if

index_a=0
   index=index_b2

!print*, 'a2, b2=', a2, b2, 'index_a2, index_b2=', index_a2, index_b2

! find the next node up the tree: note this is unique, as in jtree is a tree
! g(a,b)=0, so they can't be edge between, let alone in same clique
! hence no need to check that parent of b2 is non-zero, as it is higher
! of the two cliques: one containing a2, one containing b2

parent=0
call parents_node(jtree, index, parent)
   next_index=i_argmax(parent)
```

```
!print*, 'line 1006, index_b2-1', index_b2-1

   do dummy=1, index_b2-1
!this performs the loop at least b2-1 times if no break
         ! which must be sufficient to get to top of component
! note index_b2>=2, so does at least once
!print*, 'line 1010 inside do dummy=1 loop'

      if  ( sum(parent)==0 .AND. locate_a2 == 0 ) then
!print*, 'line 1013 inside IF sum(parent)==0 and locate_a2==0'
! if parents empty (=[0 0 ... 0]') and not found a2, must be at top of tree
         tree_top=index
         CASE_same_branch=0

!print*, 'tree tip, CASE_same_branch',tree_top,CASE_same_branch
         exit
       end if
         !if parent is empty AND locate_a2 ==0, then at the top of the tree and KNOW
         !that index_a2 and index_b2 are on separate branches. BUT can't
         !assume the fork is tree_top. could be case of 3-2-4-5 with 1-2
         !the top of tree.

clique_next_index=0
      clique_next_index=cliques(:, next_index)

!print*, 'line 1029 clique_next_index',clique_next_index

      if ( (clique_next_index(a2) ==1) .AND. (sum(parent) > 0)  ) then
         ! if a2 is in clique_next_index, and not at tree top
        index_a2=next_index
locate_a2=1
         CASE_same_branch=1
!print*, 'line 1036 locate_a2, index_a2, CASE_same_branch',locate_a2, index_a2, CASE_same_branch
exit
      end if

        ! if you find a2 before you get to the tree_top, then
        ! KNOW index_a2 and index_b2 are on same branch of tree. So
        ! break and set
        ! locate_a2 case indicator.

         index=next_index
parent=0
call parents_node(jtree, index, parent)
next_index=i_argmax(parent)
!print*, 'line 1049 index, next_index',index,next_index

    ! if a2 is in any clique on the same side of the root node in jtree, then
     ! that clique and the first clique in the RIP ordering containing b2 will be the
```

```
     ! end points of the shortest path between 2 containing cliques for the nodes of
     ! the edge considered.
    ! performs the loop at least b2-1 times if no break, which is the longest possible
    ! path to the top of a connected component of the possibly disconnected
    ! tree.

enddo

if (CASE_same_branch==0) then
    !must have locate_a2 == 0
       index_a2=0
     call find_clique_containing(a2, cliques, index_a2)
!print*, 'line 1065 index_a2',index_a2
endif

   ! IF CASE_same_branch==0, (so locate_a2==0) then
   ! index_a2 for shortest path
   ! is first clique in RIP ordering containing a2.

clique_a2_int_clique_b2_zo=0
call setintersect(cliques(:,index_a2), cliques(:,index_b2), clique_a2_int_clique_b2_zo)
s=sum(clique_a2_int_clique_b2_zo) ! size of intersection of 2 cliques
      !intersection is NOT necessarily a separator (2 cliques "far apart" in a
!connected tree can easily have empty intersection), so s is NOT
      !sepsize(index_a2, index_b2). find intersection, so can test to see if
      !it is a separator. In practice, only need to check that the size of
      !this intersection is equal to the size of a separator on this path.

!print*, 'line 1081 clique_a2_int_b2, s',clique_a2_int_clique_b2_zo, s

if (s==0) then
  edge_ok=0
new_clique=0
quit=1
!print*, 'line 1087 edge_ok, new_clique, quit', edge_ok, new_clique, quit
endif
   ! the empty set is not a separator in a connected
   ! tree, so condition cannot be satisfied. abort at this stage.

if (jtree(index_a2, index_b2)==1) then
    edge_ok=1
call setunion(ab_edge_vec, clique_a2_int_clique_b2_zo, new_clique)
quit=1
!print*, 'line 1096 edge_ok, new_clique, quit', edge_ok, new_clique, quit
endif
   !if the cliques are adjacent in the tree, their intersection is by
   !definition a separator so finished.

!!! If quit still zero, next test all the separators
!!! between clique_a2 and clique_b2 in the tree
```

```
!!! at any stage,IF there exists a separator of length s,edge_ok=1,quit=1,break,end

if ( (quit ==0) .AND. (CASE_same_branch==1) ) then
         ! first consider where a2 is on same branch as b2. only need to
         ! test path from b2 to a2
      bottom=index_b2
parent=0
call parents_node(jtree, index_b2, parent)
!print*, 'line 1112 bottom, parent', bottom, parent
if (sum(parent)==0) then
next_parent_b2=0
else
      next_parent_b2=i_argmax(parent)
endif
! if parent is empty, max(parent)=0 and i_argmax(parent)=1
! which is wrong. so must first check parent exists and
! not at top of tree component
!print*, 'line 1121 next_parent_b2', next_parent_b2

      do while( (next_parent_b2>0 ) .AND. (next_parent_b2 >= index_a2 ))
         ! while next_parents non-empty and

         if (sepsize(next_parent_b2, bottom) == s) then
           edge_ok=1
     call setunion(ab_edge_vec, clique_a2_int_clique_b2_zo, new_clique)
           quit=1
     CASE_sat_on_same_branch=1
!print*, 'line 1131 s, edge_ok, CASE_sat_on_same_branch', s, edge_ok, CASE_sat_on_same_branch
     exit
         endif
                ! only need to test equality of size,
                ! since intersection contained in every intermediate clique by RIP
        bottom=next_parent_b2

next_parent_b2=0
parent=0
call parents_node(jtree, bottom, parent)

if (sum(parent)==0) then
next_parent_b2=0
else
next_parent_b2=i_argmax(parent)
         ! next_parent_b2=find(parents_node_zo(jtree, bottom))
endif
! if parent is empty, max(parent)=0 and i_argmax(parent)=1
! which is wrong. so must first check parent exists and
! not at top of tree component
!print*, 'line 1151 bottom, next_parent_b2',bottom, next_parent_b2
```

```
      enddo;

else if ( (quit==0) .AND. (CASE_same_branch==0) ) then
!print*, 'line 1156 (quit==0) .AND. (CASE_same_branch==0)'
    ! if on different branches, have to test from index_b2 and index_a2 to
    ! fork of branch between them. CANNOT go to edge beyond fork. Safest
    ! strategy is to find the fork.

    !!! note that in below, parent clique=node of jtree is unique,
    !!  as only 1 parent in trees


! if parent is empty, max(parent)=0 and i_argmax(parent)=1
! which is wrong. so must first check parent exists and
! not at top of tree component

parent=0
call parents_node(jtree, index_a2, parent)
if (sum(parent)==0) then
next_parent_a2=0
else
      next_parent_a2=i_argmax(parent)
! using i_argmax ok as parent is unique (no need for indexes_of_nodes subroutine)
! matlab==next_parent_a2 =find(parents_node_zo(jtree, index_a2))
endif
! WHAT IS BELOW ANOTHER CALL ??? FIND OUT

parent=0
call parents_node(jtree, index_b2, parent)
if (sum(parent)==0) then
next_parent_b2=0
      else
next_parent_b2=i_argmax(parent)
endif

!! HELEN CHECK THIS IN SO FAR AS DIMENSIONS

allocate(ancestors_a2(index_a2), ancestors_b2(index_b2))
    ! already declared ancestors_a2_zo(n), ancestors_b2_zo(n)

    ancestors_a2=0 ! =zeros(1, index_a2)  ! need ancestors as a numeric NOT zero one rep
   ancestors_b2=0 ! =zeros(1, index_b2)
              !ancestors_a2 elements are actual number indicies of clique ancestors
    ancestors_a2_zo=0 ! =zeros(n,1) this is the col_vec equivalent
    ancestors_b2_zo=0 ! =zeros(n,1)

      next_ancestor_a2=index_a2
      next_ancestor_b2=index_b2

      do count=1,index_a2
```

```
!print*, 'line 1204 do count=1, index_a2: index_a2=', index_a2
       ancestors_a2(count)=next_ancestor_a2
         ancestors_a2_zo(next_ancestor_a2)=1
         parent=0
   call parents_node(jtree, next_ancestor_a2, parent)

if (sum(parent)==0) then
exit  ! i.e. if isempty(next_ancestor_a2), break, end
next_ancestor_a2=0
      else
            next_ancestor_a2=i_argmax(parent)
! using i_argmax ok as parent is unique (no need for indexes_of_nodes subroutine)
! matlab==next_ancestor_a2 =find(parents_node_zo(jtree, next_ancestor_a2))
         endif

enddo
!print*, 'line 1220 ancestors_a2=', ancestors_a2
      do count=1, index_b2
!print*, 'line 1204 do count=1, index_b2: index_b2=', index_b2
         ancestors_b2(count)=next_ancestor_b2
         ancestors_b2_zo(next_ancestor_b2)=1
parent=0
call parents_node(jtree, next_ancestor_b2, parent)

if (sum(parent)==0) then
exit  ! i.e. if isempty(next_ancestor_b2), break, end
next_ancestor_b2=0
else
next_ancestor_b2=i_argmax(parent)
  ! matlab==next_ancestor_b2 =find(parents_node_zo(jtree, next_ancestor_b2))
endif
    enddo
!print*, 'line 1236 ancestors_b2=', ancestors_b2
anc_a2_int_anc_b2_zo=0
call setintersect(ancestors_a2_zo, ancestors_b2_zo, anc_a2_int_anc_b2_zo)
! replaced fork_set by anc_a2_int_anc_b2_zo
num_int=0
num_int=sum(anc_a2_int_anc_b2_zo)

!!!! HELEN CHECK BELOW IF, AND END IT
! if (num_int>0) ! ??? i think this MUST be true, as always the top of the tree in common

allocate(anc_a2_int_anc_b2(num_int))
               !AFTERanc_a2_int_anc_b2=1, ! use for debugging
anc_a2_int_anc_b2=0
call indexes_of_nodes(anc_a2_int_anc_b2_zo, anc_a2_int_anc_b2)
! mimic matlab's anc_a2_int_anc_b2=find(anc_a2_int_anc_b2_zo==1)
! input:  1. vectorA is zero one representation of NON-ORDERED set of
!           non-zero integers. eg vectorA=[1 0 1 1 0]= a clique or separator
! output: 1. indicies is sum(vectorA) length vector of indicies where vectorA==1
```

```
!         eg find_indicies_equal_one(vectorA, indicies)=[3, 1, 4]


fork=0
  fork=maxval(anc_a2_int_anc_b2)
!print*, 'line 1259 anc_a2_int_anc_b2=', anc_a2_int_anc_b2, 'fork=', fork
      ! if no maxval use below instead: mimic fork=max(anc_a2_int_anc_b2) = actual NUMBER
! i_argmax WILL find the biggest since anc_a2_int_anc_b2 is now a node rep eg [2, 5, 6]
! fork_index=0, fork_index=i_argmax(parent), fork=anc_a2_int_anc_b2(fork_index)

                    !AFTERfork=1, ! use for debugging
              ! need to find the place in the tree where the fork is, as defined
              ! by the clique which is at this point of intersection
              ! the clique where the two paths up the tree meet must be the
              ! biggest index in the intersection to the root.
              ! i.e. if they both share clique(:,3), then the path of
              ! ancestors for both must include the ancestors of clique(:,3)
              ! which if were clique2, clique1, then where the paths split is
              ! at clique 3 = max(find(anc_a2_int_anc_b2)))

     count=1
index_row=0
index_col=0 ! initialise counters and indices

         num_seps_branch_b2=sum(ancestors_b2_zo(fork+1:n))
! num_seps_branch_b2=length(find(ancestors_b2>fork))
! in a chain (=the branch of tree), the number of separators
! is the number of cliques-1. the cliques are fork=3, 4, 7=index_last_clique_in_branch
! then num_seps=sum(0 0 0 1 1 0 1 0 0)=sum(ancestors_b2_zo(fork+1:n))

!! NOTE: num_seps_branch_b2 could =zero. CHECK FORTRAN HANDLES do 1,0
!! like matlab does, by not entering loop YES I CHECKED AND IT DOES NOTHING

!!!!  VERY IMPORTANT !!! only test separators between nodes clique_fork, and indicies
!!!!                     higher than fork i.e. further down the tree to index_b2

     do count=1, num_seps_branch_b2

index_row=ancestors_b2(count)
  index_col=ancestors_b2(count+1)
             ! sepsize is lower diagonal zero, and the ancestors are stored
             ! in descending order [index_b2,..., 1]
               ! FORTRAN version DOES have ancestors as the indicies (see above)
! as required, i.e. not a 0/1 vector

            if (sepsize( index_row, index_col) == s) then
edge_ok=1
call setunion( ab_edge_vec, clique_a2_int_clique_b2_zo, new_clique)
quit=1
CASE_sats_on_b2_branch=1
```

```
exit
            endif
      enddo
!print*, 'line 1304 num_seps_branch_b2=',num_seps_branch_b2
!print*, 'line 1305 CASE_sats_on_b2_branch',CASE_sats_on_b2_branch

if ( (quit==0 .AND. CASE_sats_on_b2_branch==0) )  then

!print*, 'line 1307 quit==0 .AND. CASE_sats_on_b2_branch==0'
            ! don't want to perform the above loop if found separator=intersection.
            !INSIDEif=1, !use to debug

num_seps_branch_a2=sum(ancestors_a2_zo(fork+1:n))
! num_seps_branch_a2=length(find(ancestors_a2>fork));
! in a chain (=the branch of tree), the number of separators
! is the number of cliques-1. the cliques are fork=3, 4, 7=index_last_clique_in_branch
! then num_seps=sum(0 0 0 1 1 0 1 0 0)=sum(ancestors_a2_zo(fork+1:n))

!print*, 'num_seps_branch_a2=', num_seps_branch_a2
do count =1, num_seps_branch_a2
index_row=ancestors_a2(count+1)
index_col=ancestors_a2(count);
             ! sepsize is lower diagonal zero, and the ancestors are stored
             ! in descending order [index_a2,..., 1]

if (sepsize( index_row, index_col) == s) then
                edge_ok=1
call setunion( ab_edge_vec, clique_a2_int_clique_b2_zo, new_clique)
                quit=1
CASE_sats_on_a2_branch=1
exit
             endif
      enddo

            !deallocate(ancestors_a2)
endif
!print*, 'line 1329 num_seps_branch_a2, CASE_sats_on_a2_branch',num_seps_branch_a2, CASE_sats_on_a2_branch
!print*, 'line 1324 deallocate next'

deallocate(ancestors_a2, ancestors_b2, anc_a2_int_anc_b2)

endif !!!!!!!!!!!!!!!!!!!!!! ENDIF the first main cases
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if (quit==0) then
  edge_ok=0
  new_clique=0
endif
```

```
END SUBROUTINE check_edge_add_same_component

!!! ------------------------------------------------------------------------

SUBROUTINE next_graph_gibbs(k_max, Ank_est_real, &
g_current, size_g_current, order, cliques, jtree, sepsize, reach_graph, &
g_next, size_g_next)

! function [g_next, size_g_next,jtree, sepsize, cliques, reach_graph]=...
!                 next_graph_gibbs(g_current, size_g_current, n, k_max, Ank_est,...
!                                  jtree, sepsize, cliques, reach_graph)

! created 25/08/04 Helen Armstrong off _zo.m equivalent
! latest update 26/08/04 (found error=ladder not initialised zero)

! generates next graph using gibbs sampler and systematic choice of ij.
! ignore draws ij outside space
! i.e. DONT count g_ij_next=g_ij when at edge ij=x, and ij=~x is not decomp
! or exceeds maximum number of edges to learn=k_max.
!  current_graph is decomposable, so doing gibbs via check edge add/delete
!  will also be decomposable

! Uses the priors of robert's notes:
!    p(g) is prop to size(g), where size(g)=num edges in g,
!    p(g|size(g))=1/An,size(g) i.e. uniform =1/num_graphs_of_size(g).

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! systematically go through all edges. Otherwise at boundary,
! q(x->y) =proposal_generating density NOT
! same as interior q=proposal_generating density, as
! nC2 poss. for interior, only k_maxC2 at boundary k_max

! NOTE: for Gibbs, ONE iteration is n*(n-1)/2 steps. Each is
! a sequential update of Jij, i=1:n, j=1:n-1

integer, intent(in) :: k_max, size_g_current
! integer, intent(inout) :: u_all_index
real(wp), intent(in), dimension(:) :: Ank_est_real  !, u_all
integer, intent(in), dimension(:,:) :: g_current
integer, intent(inout), dimension(:)   :: order
integer, intent(inout), dimension(:,:) :: cliques, jtree, sepsize, reach_graph
! NOTE: the main routine doesn't need order, or ladder, so long as it has cliques, etc
integer, intent(out) :: size_g_next
integer, intent(out), dimension(:,:) :: g_next


! local variables
integer :: i,j, index, jj
integer :: size_g_ij, skip_j
integer :: check
```

```
integer :: Jij, CASE_add, CASE_delete
integer :: accept0,  reject0

real(wp):: u, L, inv_pi_1, inv_pi_o, normalised_p

integer, dimension(n) :: old_clique_with_deletable_edge, new_clique
integer, dimension(n,n) :: g_ij, g_ij_next

!! ONLY update following if g_ij_next isn't same as the previous g_ij=g_ij
!! NOTE: for i=1, j=2 initial case, always equal, so use input to function values.
!! DO NOT CLEAR g_crnt input, as need at last few lines
!! NOTE: for Gibbs, ONE iteration is n*(n-1)/2 steps. Each is
!! a sequential update of Jij, i=1:n, j=1:n-1

g_ij=g_current
g_ij_next=g_current  ! initialise routine g_ij values
size_g_ij= size_g_current

i=0
   j=0

do i= 1, n-1
  do j=i+1, n
    skip_j=0
! indicator for outside space {size<=k_max, decomposable}
      ! skip_j=1 means "skip this j, outside space if Jij"


!!!!!! ONLY update following if g_ij_next isn't same as the previous g_ij=g_ij
!!!!!! NOTE: for i=1, j=2 initial case, always equal, so use input to function values.
   if (.not.(all(g_ij==g_ij_next))) then
     order=0
     cliques=0
     jtree=0
     sepsize=0
     reach_graph=0
     g_ij=0
     size_g_ij=0
     g_ij=g_ij_next ! update g_ij (the current graph)
     size_g_ij=sum(g_ij)/2 ! size is number of edges
     call check_chordal(g_ij, check, order)
!print*, '============================================='
!print*, 'order=', order
!print*, '============================================='
!print *, 'g_ij='
! do index = 1 , n
!     write(*,999) ( g_ij(index,jj), jj=1,n ) ! shortcut do loop
! enddo
!print*, '============================================='
```

```
     call chordal_to_ripcliques(g_ij, order, cliques) !find cliques for next_graph_candidate

!print *, 'THIS IS THE CLIQUES TO CHECK cliques='
! do index = 1 , n
!     write(*,999) ( cliques(index,jj), jj=1,n ) ! shortcut do loop
! enddo
!print*, '============================================='

call ripcliques_to_jtree(cliques, jtree)  !create jtree for next_graph_candidate
     call separator_sizes(cliques, jtree, sepsize) !find seps for next_graph_candidate
     call reachability_graph(g_ij, reach_graph)  !get connected components for n_g_cand
   endif

   !! Proposal graph must be from allowable space;
   ! i.e. have s(g)<=k_max, and decomposable
   ! i.e. only select an edge so that Jij=0 or Jij=1 is inside conditional space

    Jij=g_ij(i,j)
    CASE_add=0; CASE_delete=0;

    if ((size_g_ij==k_max) .AND. (Jij==0)) then
        skip_j=1
!print*, 'case ((size_g_ij==k_max) .AND. (Jij==0))'
         ! outside space: Jij are "edges to change in CURRENT graph". Can't add
      ! edge as size=k_max, so sampling dist. for Jij degenerate.
      ! p(Jij=1|rest)=0, p(Jij=0|rest)=1
      ! So skip this Jij parameter.
    endif

    if ( (size_g_ij==k_max) .AND. (Jij==1) ) then
!print*, 'case (size_g_ij==k_max) .AND. (Jij==1) )'
!print*, 'before re-doing CASE_delete=',CASE_delete
!print*, 'size_g_ij, cliques', size_g_ij
!print*, '============================================='
!print *, 'CHECK CASE DELETE cliques='
! do index = 1 , n
!     write(*,999) ( cliques(index,jj), jj=1,n ) ! shortcut do loop
! enddo

!print*, '============================================='
!print *, 'g_ij='
! do index = 1 , n
!     write(*,999) ( g_ij(index,jj), jj=1,n ) ! shortcut do loop
! enddo
!print*, '============================================='
       call check_edge_delete(i,j, cliques, CASE_delete, old_clique_with_deletable_edge)
!print*, 'after re-doing CASE_delete=', CASE_delete

       if (CASE_delete==1) then
!print*, 'case (size_g_ij==k_max) .AND. (Jij==1) ) AND CASE_delete==1'
```

```
        skip_j=0
!print*, 'RIGHT PLACE SKP J', skip_j
       else  ! can't delete so skip j
          skip_j=1
!print*, 'case (size_g_ij==k_max) .AND. (Jij==1) ) AND CASE_delete==0'
             ! deletion of Jij gives not decomposable, and
             ! deg. sampling distribution p(Jij=1|rest)=1, p(Jij=0|rest)=0.
             ! Can't change this Jij so don't count it as valid
             ! "next Jij" to update; i.e. skip this Jij choice
       endif
    endif

    if ((size_g_ij< k_max) .AND. (Jij==1)) then
       call check_edge_delete(i,j, cliques, CASE_delete, old_clique_with_deletable_edge)
       if (CASE_delete==1) then
          skip_j=0
!print*, 'case (size_g_ij<k_max) .AND. (Jij==1) ) AND CASE_delete==1'
       else
         skip_j=1
       endif
    endif

    if ((size_g_ij<k_max) .AND. (Jij==0)) then
!print*, 'case (size_g_ij<k_max) .AND. (Jij==0)'
       if (reach_graph(i,j)==0) then
          skip_j=0
       endif
             !!!  NO NEED TO CHECK CASE ADD.
             ! if i and j are in different connected components of a decomposable
             ! graph, then new graph containing edge between them is also
             ! decomposable.
             ! if adding edge a-b between 2 disjoint trees, new clique
             ! containing that edge must be comprised soley of edge nodes a,b

       if (reach_graph(i,j)==1) then
          call check_edge_add_same_component(i,j, jtree, sepsize, cliques, CASE_add, new_clique)
                   ! i,j must be in same connected component of
                   ! tree for check_edge_add.m

          if (CASE_add /= 1) then
              skip_j=1
          endif
       endif
    endif ! if (size_g_ij<k_max & Jij==0)

   !!!!!!!!! Only randomly select Jij if skip_j==0
   !!!!!!!!! Else just ignore this Jij and move to next j
   !!!!!!!!! Will do GIBBS based on cdf for Jij=0, Jij=1 order, so only test
   !!!!!!!!!  u vs p(Jij=0) line (see notes in MCMC GIBBS stuff
```

```
   !!!!!!!!! Test is based on size of CURRENT graph where you ignore Jij itself,
   !!!!!!!!! Then pi_o, pi_1 depend on size(g_ij with Jij=0), and size(g_ij with Jij=1)

   !!! NOTE to helen: it's ok not to check for "if case delete =, if add=" because
   !!! we know (because skip_j=0) that Jij can be either 0 or 1. So either is decomposable,
   !!! and within space size <=k_max. Hence, so long as know size of current graph, can
   !!! work out bivariate distribution from which to randomly select Jij.

!print*, 'test to see if skip j=', skip_j
      if (skip_j==0) then
!print*, 'case (skip_j==0)'

           if (Jij==1) then  ! Jij=g_ij(i,j)
             inv_pi_o=Ank_est_real(size_g_ij-1+1)
             inv_pi_1=Ank_est_real(size_g_ij+1)
                    ! Phi=Phi_o, Phi_1,.. so size=k is Phi(1,k+1)
                    ! Currently Jij=1
                    ! So UNNORMALISED
                    ! pi_0 of Jij=0|rest=1/(Phi(size(g_ij)-1) +1)
                    ! pi_1= of Jij=1|rest=1/(Phi(1, size(g_ij) +1).
                    ! i.e if g_ij(i,j) currently 1,
                    !     p(Jij=0)=p(g_Jij,rest)=p('g_ij with Jij deleted')
                    ! but p(Jij=1)=p(g_Jij,rest)=p(g_ij) as g_ij(i,j) currently=Jij

            else ! Jij=0 already is only other possibility
      inv_pi_o=Ank_est_real(size_g_ij+1)
      inv_pi_1=Ank_est_real((size_g_ij+1)+1)
            ! Currently Jij=0.
            ! pi for Jij=0|rest=1/Phi(size(g_ij) +1).
            ! pi for Jij=1|rest=1/(Phi(1,size(g_ij)+1+1),
            ! since g_Jij,rest=g_ij + extra edge Jij=1 if g_ij(i,j)=0, etc

    endif ! if Jij==1/0


    !!!!!! Now normalise bivariate pi_o, pi_1 via
    !!!!!! normalised =pi_o/(pi_o+pi_1), =pi_1/(pi_o+pi_1).
    ! Then can randomly draw the next Jij from this bivariate
    ! distribution. Note that the u test below is not metropolis
    ! hastings test, but just a way to draw from the bivariate
    ! distribution normalised_p, a single normalised_p based on pi_o via
    ! it's cdf graph. (Gibbs actually=MH with q(x,y) prop p(x,y), a
    ! symmetric MH density that satisfies reversibility (see Chibb, Greenberg))

    !! working precision pi_o=1/Ank-1 ~0, so
    !! use L=pi_0/pi_1=inv_pi_1/inv_pi_o=A_nk/A_nk-1 (actually use Phi guesses)
    ! Ank's only one apart should be of same order.
    ! hence L is close to 1, or at least order .01-100
    ! so DON'T take logs as log 1=0.
```

```
            ! might need to do cases using inv(L) for test
            ! i.e. do gibbs on case pi_1 rather than pi_0
            ! refer ed's program

      call r_uniform(u)
! u_all_index=u_all_index+1 for debugging, use a fixed data set of uniform

! u=u_all(u_all_index)
  normalised_p=0.0_wp
           L=inv_pi_1/inv_pi_o ! = pi_o/pi_1, so TEST BASED on Jij=0 ie Ank/Ank-1
           normalised_p=L/(1+L) ! = prob(Jij=1|rest, decomposable, size<=k_max)

           if (u < normalised_p) then
                accept0=1
                g_ij_next=g_ij
                g_ij_next(i,j)=0
                g_ij_next(j,i)=0
! print*, '[i,j], Jij, accept0', Jij, accept0, (/i, j/)

           else  ! Jij bivariate=0/1. if not zero, must be 1 (gibbs)
                reject0=1
                g_ij_next(i,j)=1
                g_ij_next(j,i)=1
!print*, '[i,j], Jij, reject0', Jij, reject0, (/i,j/)
           endif

!print*, 'u, u_all_index, L, normalised_p', u, u_all_index, L, normalised_p
        endif ! if skip_j==0
    enddo ! for j loop
enddo ! for i loop

!!!! end GIBBS generate g_next via updating all Jij sequentially
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! the final output is the last graph you're at after all updates
! if it's not same as g_current inputted, then update all inputs

! NOTE: you CANNOT do test g_ij, g_ij_next as these might be the same,
! but the input g_current may not be the same as the current g_ij_next

g_next=g_ij_next ! the final output is the last graph you're at after all updates
size_g_next=sum(g_next)/2
if (.not.(all(g_current== g_next))) then  ! CANNOT do test g_ij, g_ij_next (see above)
cliques=0; jtree=0; order=0;
sepsize=0; reach_graph=0;
call check_chordal(g_next, check, order)
call chordal_to_ripcliques(g_next, order, cliques)
call ripcliques_to_jtree(cliques, jtree)
call separator_sizes(cliques, jtree, sepsize)
call reachability_graph(g_next, reach_graph)
endif
```

```
!print*, 'size_g_next', size_g_next
!print*, '============================================='
!print *, 'g_next='
! do index = 1 , n
!     write(*,999) ( g_next(index,jj), jj=1,n ) ! shortcut do loop
! enddo
!print*, '============================================='
!print*, 'skip_j=', skip_j
!print*, '============================================='

!999 format(25I5)  ! make the format 20 integers, each I=integer is 5 spaces
END SUBROUTINE next_graph_gibbs
!!! ------------------------------------------------------------------------
END MODULE graph_mod
```

Bibliography

Atay-Kayis, A. & Massam, H. (2005). A Monte Carlo method to compute the marginal likelihood in non decomposable graphical gaussian models. *Biometrika* in press.

Barnard, J., McCulloch, R., & Meng, X. (2000). Modeling covariance matrices in terms of standard deviations and correlations, with application to shrinkage. *Statistica Sinica* **10**, 1281–1311.

Brooks, S., Giudici, P., & Roberts, G. O. (2003). Efficient construction of reversible jump Markov chain Monte Carlo proposal distributions (with discussion). *J. Royal Statistical Society B* **65**, 3–55.

Brown, P., Fearn, T., & Vannucci, M. (1999). The choice of variables in multivariate regression: a non-conjugate Bayesian decision theory approach. *Biometrika* **86**, 635–648.

Brown, P., Vannucci, M., & Fearn, T. (1998). Multivariate Bayesian variable selection and prediction. *Journal of the Royal Statistical Society, Series B* **60**, 627–641.

Brown, P., Vannucci, M., & Fearn, T. (2002). Bayes model averaging with selection of regressors. *Journal of the Royal Statistical Society, Series B* **64**, 519–536.

Castelo, R. & Wormald, N. (2001). Enumeration of p4-free chordal graphs. *Journal of Graphs and Combinatorics* (in press), or Universiteit Utrecht, Technical Report UU-CS-2001-12, June 2001.

Chiu, T., Leonard, T., & Tsui, K. (1996). The matrix-logarithm covariance model. *Journal of the American Statistical Association* **81**, 310–20.

Cripps, E., Carter, C., & Kohn, R. (2005). Variable selection and covariance selection for multivariate regression models. In Dey, D. & Rao, C., editors, *Handbook of Statistics 25: Bayesian Thinking: Modeling and Computation*, chapter 18, pages 519–552. Elsevier B. V., Amsterdam.

Dawid, A. (1979). Conditional independence in statistical theory. *J. Royal Statistical Society B* .

Dawid, A. (1981). Some matrix-variate distribution theory: notational considerations and a Bayesian application. *Biometrika* **68**, 265–274.

Dawid, A. P. & Lauritzen, S. (1993). Hyper Markov laws in the statistical anlaysis of decomposable graphical models. *The Annals of Statistics* **21**, 1272–1317.

Dellaportas, P. & Forster, J. (1999). Markov chain Monte Carlo model determination for heirachical and graphical log-linear models. *Biometrika* **86**, 615–633.

Dellaportas, P., Giudici, P., & Roberts, G. (2004). Bayesian inference for non-decomposable graphical Gaussian models. Sankyha, Series A.

Dempster, A. (1972). Covariance selection. *Biometrics* **28**, 157–175.

Dempster, A. P. (1969). *Elements of Continuous Multivariate Analysis*. Addison-Wesley, Reading, MA.

Diggle, P., Heagerty, P., Liang, K., & Zeger, S. (2002). *Analysis of longitudinal data*. Oxford University Press.

Drton, M. & Perlman, M. D. (2004). Model selection for Gaussian concentration graphs. *Biometrika* **91**, 591–602.

Efron, B. & Morris, C. (1976). Multivariate Empirical Bayes estimation of covariance matrices. *Annals of Statistics* **4**, 22–32.

Frydenberg, M. & Lauritzen, S. (1989). Decomposition of maximum likelihood in mixed interaction models. *Biometrika* **76**, 539–555.

Geiger, D. & Heckerman, D. (2002). Parameter priors for directed acyclic graphical models and the characterization of several probability distributions. *Annals of statistics* **30**, 1412–1440.

Gelman, A., Carlin, J., Stern, H., & Rubin, D. (2000). *Bayesian Data Analysis*. Chapman and Hall/CRC.

Giudici, P. (1996). Learning in graphical Gaussian models. In J. Berger, J. M. Bernardo, A. P. D. & Smith, A. F. M., editors, *Bayesian Statistics 5: Proceedings of the Fifth Valencia International Meeting, June 5-9, 1994*, pages 621–628. Oxford University Press.

Giudici, P. & Castelo, R. (2003). Improving Markov chain Monte Carlo model search for data mining. *Machine learning* **50**, 127–158.

Giudici, P. & Green, P. J. (1999). Decomposable graphical Gaussian model determination. *Biometrika* **86**, 785–801.

Grone, R., Johnson, C., Sa, E., & Wolkowice, H. (1984). Positive definite completions of partial Hermitian matrices. *Linear Algebra Applications* **58**, 109–124.

Hammersley, J. M. & Clifford, P. E. (1971). Markov fields on finite graphs and lattices. Unpublished manuscript.

Jones, B., Carvalho, C., Dobra, A., Hans, C., Carter, C., & West, M. (2005). Experiments in stochastic computation for high-dimensional graphical models. *Statistical Science* **20**, 388–400.

Kalmanson, K. (1986). *An introduction to discrete mathematics and applications*. Addison-Wesley Publishing Company.

Kohn, R., Smith, M., & Chan, D. (2001). Nonparametric regression using linear combinations of basis functions. *Statistics and Computing* **11**, 313–322.

Larner, M. (1996). Mass and its relationship to physical measurements. Technical Report MS305 Data Project, Department of Mathematics, University of Queensland. The data can be downloaded from http://www.statsci.org/data/oz/physical.html.

Lauritzen, S. L. (1996). *Graphical models*. Oxford University Press.

Leimer, H.-G. (1989). Triangulated graphs with marked vertices. In L.D. Andersen, C. Thomassen, B. T. & (ed.), P. V., editors, *Graph Theory in Memory of G. A. Dirac*, volume 41 of *Annals of Discrete Mathematics*. Elsevier Science Publishers B.V. (North Holland), Amsterdam.

Liechty, J. C., Liechty, M. W., & Müller, P. (2004). Bayesian correlation estimation. *Biometrika* **91**, 1–14.

Mardia, K. V., Kent, J. T., & Bibby, J. M. (1979). *Multivariate Analysis*. London: Academic Press.

Muirhead, R. (1982). *Aspects of Multivariate Statistical Theory*. Wiley.

ODELL, P. & FEIVESON, A. (1966). A numerical procedure to generate a sample covariance matrix. *Journal of the American Statistical Association* .

PAULSEN, V. I., POWER, S. C., & SMITH, R. R. (1989). Schur products and matrix completion. *Journal of Functional Analysis* **85**, 151–78.

ROVERATO, A. (2000). Cholesky decomposition of a hyper inverse Wishart matrix. *Biometrika* **87**, 99–112.

ROVERATO, A. (2002). Hyper inverse Wishart distribution for non-decomposable graphs and its application to Bayesian inference for Gaussian graphical models. *Scandinavian Journal of Statistics* **29**, 391–411.

ROVERATO, A. & WHITTAKER, J. (1998). The isserlis matrix and its application to non-decomposable graphical Gaussian models. *Biometrika* **85**, 711–725.

SMITH, M. & KOHN, R. (1996). Nonparametric regression using bayesian variable selection. *Journal of Econometrics* **75**, 317–342.

SMITH, M. & KOHN, R. (2002). Bayesian parsimonious covariance matrix estimation for longitudinal data. *Journal of the American Satistical Association* **87**, 1141–1153.

SPEED, T. & KIIVERI, H. T. (1986). Gaussian Markov distributions over finite graphs. *Annals of Statistics* **14**, 138–150.

SUNDBERG, R. (1975). Some results about decomposable (or Markov-type) models for multidimensional contingency tables: Distribution of marginals and partitioning of tests. *Scandinavian journal of statistics* **2**, 71–79.

TARJAN, R. & YANNAKAKIS, M. (1984). Simple linear time algoithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM journal of computing* .

WERMUTH, N. (1976). Analogies between multiplicative models in contingency tables and covariance selection. *Biometrics* **32**, 95–108.

WERMUTH, N. (1980). Linear recursive equations, covariance selection, and path analysis. *Journal of the American Statistical Association* **75**, 963–972.

WONG, F., CARTER, C., & KOHN, R. (2003). Efficient estimation of covariance selection models. *Biometrika* **90**, 809–830.

Wong, K. F. K. (2002). An efficient sampler for decomposable covariance selection model. Master's thesis, The Hong Kong University of Science and Technology.

Wormald, N. (1985). Counting labelled chordal graphs. *Graphs and combinatorics* **1**, 193–200.

Yang, R. & Berger, J. (1994). Estimation of a covariance matrix using the reference prior. *Annals of Statistics* **22**, 1195–1211.