NORGES TEKNISK-NATURVITENSKAPELIGE
UNIVERSITET

`GMRFLib`: a `C`-library for fast and exact simulation of Gaussian Markov random fields

by

Håvard Rue & Turid Follestad

PREPRINT
STATISTICS NO. 1/2002

NORWEGIAN UNIVERSITY OF SCIENCE AND
TECHNOLOGY
TRONDHEIM, NORWAY

This report has URL `http://www.math.ntnu.no/preprint/statistics/2002/S1-2002.ps`
Håvard Rue has homepage: `http://www.math.ntnu.no/~hrue`
E-mail: `hrue@math.ntnu.no`
Address: Department of Mathematical Sciences, Norwegian University of Science and Technology, N-7491 Trondheim, Norway.

GMRFLib: a C-library for fast and exact simulation of Gaussian Markov random fields

Version 1.0

Håvard Rue & Turid Follestad

Department of Mathematical Sciences
The Norwegian Institute of Technology
N–7491 Trondheim, Norway.
E–MAIL: {havard.rue,turid.follestad}@math.ntnu.no
URL: http://www.math.ntnu.no/~hrue/GMRFLib

December 2001

Abstract

This manual describes the library GMRFLib of C-routines for fast and exact simulation of Gaussian Markov Random Fields (GMRF) on graphs. The library performs

- unconditional simulation of a GMRF,
- various types of conditional simulation from a GMRF,
- evaluation of the corresponding log-density, and
- generation of blockupdates in MCMC-algorithms.

Copyright

GMRFLib: Fast and exact simulation of Gaussian Markov random fields on graphs.

Copyright © 2001 Håvard Rue

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Version

```
$Id: GMRFLib-manual.tex,v 1.6 2002/01/06 19:49:48 hrue Exp hrue $
```

ChangeLog

Version 1.0, December 2001 This version is a complete rewrite of and older library, `GMRFsim`. New features are added, the internal data-structures are new, the user-interface is very much simplified. A new manual is written.

Contents

1 Introduction

This manual gives the documentation of the library `GMRFLib` of routines for fast and exact simulation of Gaussian Markov random fields on graphs, including the special cases of two-dimensional lattice graphs and one-dimensional linear graphs. The algorithms are computationally fast, and in most cases with n variables or nodes in the graph, the algorithms require $\mathcal{O}(n)$ flops in the one-dimensional case, and $\mathcal{O}(n^2)$ flops in the two-dimensional case. The requirement is that the bandwidth b_w of the precision matrix of the Gaussian Markov random field (GMRF) is relatively small compared to n, $\mathcal{O}(1)$ in the one-dimensional case and $\mathcal{O}(\sqrt{n})$ in the two-dimensional case. The bandwidth requirement is after a possible reordering of the node. For most problems, there exists a reordering of the nodes of the graph of the problem that results in a precision matrix with a relatively small bandwidth. This fact is the basis of the implementation of the routines in `GMRFLib`. As an example, consider spatial applications, where a variable is observed in a set of sub-regions, spanning a larger region, e.g. a country. It will often be the case that neighbouring regions are similar, but that regions further apart will be conditionally uncorrelated. Thus, there will exist a re-ordering the regions such that the precision matrix has a bandwidth that is small compared to the number of regions.

In the next section, we give a brief introduction to Gaussian Markov random fields. In Section 3, a description of the different tasks solved by the `GMRFLib`-library, as well as some background on the implementation of the different routines are given. A detailed documentation of the storage schemes and `c`-routines is given in Section 4, and Section 5 contain some examples of how to use the routines, including a worked out example form spatial disease mapping. In Section 6, instructions on how to install and compile the code are given.

2 A brief introduction to Gaussian Markov random fields

All routines in `GMRFLib` are computations on Gaussian Markov Random Fields (GMRF) defined on graphs. Before defining a general graph, we describe a GMRF on a lattice, which is a special case of a graph.

Let Λ be a two-dimensional lattice with size $n_{\text{row}} \times n_{\text{col}}$, which we index by $i = 0, \ldots, n_{\text{row}} - 1$ and $j = 0, \ldots, n_{\text{col}} - 1$. Further, let $\boldsymbol{x}$ be a Gaussian field (GF) with mean $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, defined on the lattice Λ. Then, $\boldsymbol{x}_{ij}$ is the value of the GF $\boldsymbol{x}$ at lattice site $(i,j);\ i = 0, \ldots, n_{\text{row}} - 1;\ j = 0, \ldots, n_{\text{col}} - 1$.

The correlation structure of a Gaussian Markov random field (GMRF) is usually defined by the precision matrix $\boldsymbol{Q} = \boldsymbol{\Sigma}^{-1}$ rather than the covariance matrix $\boldsymbol{\Sigma}$. The structure of $\boldsymbol{Q}$ is defined by the *neighbourhood* system. Denote by ∂_{ij} the neighbourhood of site (i,j). The library `GMRFLib` uses a *rectangular window* to define the neighbourhood ∂_{ij} of (i,j), such that

$$\partial_{ij} = \{(k,l) \in \Lambda \ :\ |i-k| \le m_{\text{row}} \quad \text{and} \quad |j-l| \le m_{\text{col}}\}. \tag{1}$$

Here, m_{row} and m_{col} are constants and do not depend on (i,j). This choice of ∂_{ij} defines the neighbourhood of (i,j) as the $(2m_{\text{row}} + 1) \times (2m_{\text{col}} + 1)$ window centred at (i,j). Common choices are $m_{\text{row}} = m_{\text{col}} = 1, 2$, and 3, giving a neighbourhood system of size 3×3, 5×5 or 7×7. For a further discussion, see (Rue and Tjelmeland, 2002).

A GMRF $\boldsymbol{x}$ is a Gaussian field for which the following Markov property hold: x_{ij} and x_{kl} are conditional independent given $\boldsymbol{x}_{\Lambda \setminus \{(i,j),(k,l)\}};\ \forall (k,l) \notin \partial_{ij}$. This could also be stated by $\pi(x_{ij} | \boldsymbol{x}_{-ij}) = \pi(x_{ij} | \boldsymbol{x}_{\partial_{ij}})$. The precision matrix $\boldsymbol{Q}$ of a GMRF has the property that $Q_{ij,kl} = 0$ if $(k,l) \notin \partial_{ij}$. Hence, only a few terms in $\boldsymbol{Q}$ have to be specified. For each $(i,j) \in \Lambda$, these elements are $Q_{ij,kl}$ for all $(k,l) \in \partial_{ij}$ and the diagonal element $Q_{ij,ij}$.

Note that $\boldsymbol{Q}$ must be positive definite and symmetric.

A *general* graph is defined by a set of nodes Λ, not necessarily on a lattice, and a neighbourhood, ∂_i; $\forall i \in \Lambda$, of nodes, defined such that for a Gaussian field $\boldsymbol{x}$, the conditional density of x_i given $\boldsymbol{x}_{-i}$ only depends on $\boldsymbol{x}_{\partial_i}$. This constitutes the Markov property of a GMRF on a general graph. A lattice is a special case of graph, where the nodes are regularly spaced, and the neighbourhood has a regular form.

3 Description of the tasks solved by `GMRFLib`

The `GMRFLib` library contains routines for solving three main tasks concerning GMRF's. These are unconditional and conditional sampling from a GMRF, and evaluation of the log-density of a sample from a GMRF. The routines are implemented for a general subset of a GMRF $\boldsymbol{x}$ for which the density can be written in the general form

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T(\boldsymbol{Q}+\operatorname{diag}(\boldsymbol{c}))(\boldsymbol{x}-\boldsymbol{\mu}) + \boldsymbol{b}^T\boldsymbol{x}\right). \tag{2}$$

The mean and covariance matrix of $\boldsymbol{x}$ are given by

$$\mathrm{E}(\boldsymbol{x}) = \boldsymbol{\mu} + (\boldsymbol{Q}+\operatorname{diag}(\boldsymbol{c}))^{-1}\boldsymbol{b}, \qquad \text{and} \tag{3}$$

$$\operatorname{Var}(\boldsymbol{x}) = (\boldsymbol{Q}+\operatorname{diag}(\boldsymbol{c}))^{-1}. \tag{4}$$

The conditioning take the general form of either

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e} \tag{5}$$

or

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e} + \boldsymbol{\epsilon} \tag{6}$$

where $\boldsymbol{\epsilon}$ is a zero mean Gaussian variable with covariance matrix $\boldsymbol{\Sigma}$, and $\boldsymbol{A}$ is a $k \times n$ matrix of rank k. The conditioning is computed in an efficient way, requiring $\mathcal{O}(k^3)$ additional flops.

In this section, we describe the tasks that can be solved using `GMRFLib`, referring to the objects and routines that are to be used. A detailed description and documentation of the objects and routines are given in Section 4. All routines are based on the operations on graphs, and before turning to a description of the sampling algorithms, we describe how to specify and handle graphs.

3.1 Specification and handling of graphs

The GMRF can be specified on a general graph or on a regular lattice. In general, a graph is defined by a set of n nodes, and the set of neighbours, denoted δ_i, for each node i. In `GMRFLib`, all graphs are represented by the same data structure, described in Section 4.1.1. The user can specify the neighbourhood structure of a graph by creating and initializing an instance of this data structure in a `C`-program. However, the simplest approach to specifying a general graph within `GMRFlib` is reading the specification from a file.

For a few special cases, the graphs can alternatively be specified using type-specific routines, specifying the parameters defining the graphs. Whatever method is used to specify these graphs, the graphs will be represented by the general data structure. The functions are introduced to simplify the specification of these graphs, since the neighbourhood structures are specified by a limited number of parameters. The special cases that are supported by separate routines are lattice graphs, linear graphs and the graph corresponding to weighted averages (wa-graphs).

- A *regular lattice graph* can be represented by a set of nodes on a n_{row} by n_{col} lattice, such that $n = n_{\text{row}} n_{\text{col}}$. In the case of a lattice, the neighbourhood δ_i of node i is defined as a rectangular window of size $2m_{row}+1$ by $2m_{col}+1$, around the centre point i, as described in Section 2. The graph is fully specified by the parameters n_{row}, n_{col}, m_{row} and m_{col}.

- A *linear graph* refers to one-dimensional graphs representing auto-regressive (AR) models of order p. The linear graph represents the conditional independence structure of a GMRF having density of the form

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{\kappa}{2}\sum_i\left(x_i - \sum_{k=1}^{p}\phi_k x_{i-k}\right)^2\right), \tag{7}$$

  letting $\mathrm{E}(\boldsymbol{x}) = \mathbf{0}$.

- The *wa-graph* is introduced as the corresponding graph to a density of the form

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{1}{2}\sum_i\left(w_{ii}x_i - \sum_{j\sim i} w_{ij}x_j\right)^2\right). \tag{8}$$

  The weights are not required to be symmetric, such that in general $w_{ij} \neq w_{ji}$. The reason for treating this model separately, is that by specifying a GMRF with density of the form (8), it is often easier and more intuitive to specify the weights w_{ij} and their corresponding graph, than specifying the neighbourhood structure of the GMRF $\boldsymbol{x}$ directly, using the representation (2). (However: neither one is easy!) In `GMRFLib`, a routine `GMRFLib_init_wa_problem` is provided for transforming from the representation (8) to the representation (2).

The most important operations available concerning specification and handling of graphs and lattices, along with the corresponding functions, are listed in Table 3.1.

Operation	*Function*
Read a graph from a file	`GMRFLib_read_graph`
Print the specification of a graph to standard output or a file	`GMRFLib_print_graph`
Create an empty graph	`GMRFLib_create_graph`
Free the memory held by a graph	`GMRFLib_free_graph`
Create a graph on a lattice	`GMRFLib_create_lattice`
Create the graph of a linear autoregressive AR(p)-model	`GMRFLib_make_linear_graph`
Extract a subgraph of a graph	`GMRFLib_compute_subgraph`
Check the graph definition for consistency	`GMRFLib_validate_graph`
Prepare a user-defined graph for computations	`GMRFLib_prepare_graph`
Combine two graphs to create an expanded graph	`GMRFLib_fold_graph`
Expand a graph by multiple folding (g $*$ g $*\cdots$)	`GMRFLib_nfold_graph`
Define the ordering of the nodes of a lattice	`GMRFLib_lattice2node`
Create a lattice from a set of ordered nodes	`GMRFLib_node2lattice`
Check whether nodes are neighbours	`GMRFLib_is_neighb`
Compute $\boldsymbol{Qx}$ for the precision matrix $\boldsymbol{Q}$ and a vector $\boldsymbol{x}$	`GMRFLib_Qx`
Remove nodes of a graph for which $Q(i,j)=0$	`GMRFLib_prune_graph`
Convert the graph of a weighted average problem to a general graph	`GMRFLib_init_wa_problem`

Table 1: Functions operating on graphs.

3.2 Sampling from a GMRF

The `GMRFLib` is designed for solving the general problem of sampling from a GMRF $\boldsymbol{x}$ of size n with density given by the general expression (2). The library supports unconditional simulation and three types of conditional simulation: conditioning on fixed values, on a deterministic linear constraint or on a linear constraint observed with noise. Formally, the possible constraints are

$$x_i;\ i \in \mathcal{F} \text{ are fixed} \tag{9}$$

and

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e} + \boldsymbol{\epsilon},\ \text{where} \begin{cases} \boldsymbol{\epsilon} \equiv \boldsymbol{0} & \text{(deterministic linear constraint)} \\ \boldsymbol{\epsilon} \sim N(\boldsymbol{0}, \boldsymbol{\Sigma}_\epsilon) & \text{(stochastic linear constraint)} \end{cases} \tag{10}$$

Sampling from (2) conditionally on a stochastic linear constraint, is equivalent to sampling from the unconditional distribution

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T(\boldsymbol{Q}+\text{diag}(\boldsymbol{c}))(\boldsymbol{x}-\boldsymbol{\mu}) + \boldsymbol{b}^T\boldsymbol{x} - \frac{1}{2}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{e})^T\boldsymbol{\Sigma}_\epsilon^{-1}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{e})\right). \tag{11}$$

All these simulation methods are implemented in one routine, `GMRFLib_sample`, documented in Section 4.3.

The library also includes a routine for constructing block-sampling schemes in MCMC algorithms, `GMRFLib_blockupdate`, sampling from the general model

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T(\boldsymbol{Q}+\text{diag}(\boldsymbol{c}))(\boldsymbol{x}-\boldsymbol{\mu}) + \boldsymbol{b}^T\boldsymbol{x} + \sum d_i f_i(x_i)\right). \tag{12}$$

(under the same general constraints as above). Here, the functions $f_i(x_i)$ are general functions of the components of the GMRF $\boldsymbol{x}$.

In the subsections to follow, we describe the sampling methods in more detail.

3.2.1 Unconditional simulation

Unconditional simulation from a GMRF $\boldsymbol{x}$ with distribution given by (2) can be done fast and exact. The sampling routines utilize the fact that by computing the band-Cholesky factorization

$$\boldsymbol{Q} = \boldsymbol{L}\boldsymbol{L}^T \tag{13}$$

and then solving by band back-substitution

$$\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{z} \tag{14}$$

where $\boldsymbol{z}$ is a vector of $i.i.d$ standard Gaussian variables, $\boldsymbol{x}$ is an exact sample from a GMRF with precision matrix $\boldsymbol{Q}$ and zero mean (Rue, 2001). Further, it is assumed that $\boldsymbol{Q}$ is sparse and banded with a relatively small bandwidth, b_w. The graphs on which the GMRF is defined, are customized such that the bandwidth is minimized for the problem at hand. As an example, consider a $n_{\text{row}} \times n_{\text{col}}$ lattice graph with a neighbourhood defined by a $2m_{row}+1$ by $2m_{col}+1$ window The bandwidth b_w is chosen to be

$$b_w = \min\{b_{w1}, b_{w2}\}, \tag{15}$$

where

$$\begin{aligned} b_{w1} &= \min\{m_{\text{col}}, n_{\text{col}}-1\} \times n_{\text{row}} + \min\{m_{\text{row}}, n_{\text{row}}-1\} \text{ and} \\ b_{w2} &= \min\{m_{\text{row}}, n_{\text{row}}-1\} \times n_{\text{col}} + \min\{m_{\text{col}}, n_{\text{col}}-1\}. \end{aligned} \tag{16}$$

The bandwidths b_{w1} and b_{w2} correspond to different storage schemes, column-wise and row-wise respectively, for which the most efficient one is chosen in `GMRFLib`. The matrix $\boldsymbol{L}$ can be computed using only $n_{\text{row}} n_{\text{col}} b_w^2$ flops, and $\boldsymbol{Lx} = \boldsymbol{z}$ can be solved using only $2n_{\text{row}} n_{\text{col}} b_w$ flops (Golub and van Loan, 1996). High-quality routines for computing (13) and (14) are available in the `Lapack` Fortran-library (Anderson et al., 1995).

For a general graph, if the bandwidth of the precision matrix based on the original ordering of the nodes is not small, the `GMRFLib`-library will re-order the nodes so as to minimize the bandwidth and thus increase the efficiency of the sampling.

Once the band-Cholesky factorization $\boldsymbol{L}$ has been computed, repeated samples can be generated by solving (14) for a set of samples $\boldsymbol{z} \overset{iid}{\sim} N(\boldsymbol{0}, \boldsymbol{I})$, reusing $\boldsymbol{L}$.

3.3 Conditional simulation

3.3.1 Conditioning on a linear constraint

A conditional sample $\boldsymbol{x}^*$ from the conditional distribution $\boldsymbol{x}|\boldsymbol{Ax} = \boldsymbol{e}$, where $\boldsymbol{A}$ is a $n_{\text{cond}} \times n$ matrix of rank n_{cond}, and $\boldsymbol{e}$ is a vector of length n_{cond}, is obtained from an unconditional sample $\boldsymbol{x}$, by calculating

$$\boldsymbol{x}^* = \boldsymbol{x} - \boldsymbol{Q}^{-1}\boldsymbol{A}^T \left(\boldsymbol{AQ}^{-1}\boldsymbol{A}^T\right)^{-1} (\boldsymbol{Ax} - \boldsymbol{e})\,. \tag{17}$$

This is commonly referred to as conditioning using Kriging, see Cressie, 1993, Sec. 3.6.2. If repeated conditional samples are required, then some of the calculations in (17) can be reused. `GMRFLib` takes advantage of this by reusing

$$\boldsymbol{Q}^{-1}\boldsymbol{A}^T \left(\boldsymbol{AQ}^{-1}\boldsymbol{A}^T\right)^{-1}. \tag{18}$$

3.3.2 Conditioning on a stochastic linear constraint

A sample $\boldsymbol{x}^*$ from the conditional distribution $\boldsymbol{x} \mid \boldsymbol{Ax} + \boldsymbol{\epsilon} = \boldsymbol{e}$, where $\boldsymbol{A}$ and $\boldsymbol{e}$ are defined as in Section 3.3.1 and $\boldsymbol{\epsilon} \sim N(\boldsymbol{0}, \boldsymbol{\Sigma}_\epsilon)$, can be obtained in a similar way as in the case of a deterministic constraint, with a slight modification of (17). From an unconditional sample $\boldsymbol{x}$, a conditional sample $\boldsymbol{x}^*$ is obtained by sampling an $\boldsymbol{\epsilon}^* \sim N(\boldsymbol{0}, \boldsymbol{\Sigma}_\epsilon)$, and then computing

$$\boldsymbol{x}^* = \boldsymbol{x} - \boldsymbol{Q}^{-1}\boldsymbol{A}^T \left(\boldsymbol{AQ}^{-1}\boldsymbol{A}^T + \boldsymbol{\Sigma}_\epsilon\right)^{-1} (\boldsymbol{Ax} + \boldsymbol{\epsilon}^* - \boldsymbol{e})\,. \tag{19}$$

To prove (19), compute the mean and covariance matrix of $\boldsymbol{x}^*$, and compare it to the values obtained using multivariate normal theory, which are

$$\begin{aligned} \mathrm{E}(\boldsymbol{x}^*) &= \mathrm{E}(\boldsymbol{x}) - \boldsymbol{Q}^{-1}\boldsymbol{A}^T \left(\boldsymbol{AQ}^{-1}\boldsymbol{A}^T + \boldsymbol{\Sigma}_\epsilon\right)^{-1} (\boldsymbol{A}\,\mathrm{E}(\boldsymbol{x}) - \boldsymbol{e})\,, && (20)\\ \mathrm{Var}(\boldsymbol{x}^*) &= \left(\boldsymbol{Q} + \boldsymbol{A}^T\boldsymbol{\Sigma}_\epsilon^{-1}\boldsymbol{A}\right)^{-1}. && (21) \end{aligned}$$

The comments on computational efficiency made in Section 3.3.1 apply to the case of a stochastic constraint as well.

3.3.3 Conditioning on fixed values

Assume that the GMRF $\boldsymbol{x}$ is observed for a subset $\mathcal{F}$ of the nodes in the set Λ, and denote the observed values by $\boldsymbol{x}_{\mathcal{F}}$. Similarly let $-\mathcal{F}$ denote the remaining nodes, that is $-\mathcal{F} = \{i : i \in$

$\Lambda \backslash \mathcal{F}\}$. We want to sample from the conditional distribution of $\boldsymbol{x}_{\mathcal{F}}$ given $\boldsymbol{x}_{-\mathcal{F}}$, $\pi(\boldsymbol{x}_{\mathcal{F}} \mid \boldsymbol{x}_{-\mathcal{F}})$. Let $\boldsymbol{x}_1 = \boldsymbol{x}_{\mathcal{F}}$ and $\boldsymbol{x}_2 = \boldsymbol{x}_{-\mathcal{F}}$, and partition the precision matrix, $\boldsymbol{Q}$, it's inverse $\boldsymbol{Q}^{-1}$ and $\boldsymbol{\mu}$ and $\boldsymbol{b}$ in (2) accordingly, such that

$$\boldsymbol{Q} = \begin{pmatrix} \boldsymbol{Q}_{11} & \boldsymbol{Q}_{12} \\ \boldsymbol{Q}_{21} & \boldsymbol{Q}_{22} \end{pmatrix}, \quad \boldsymbol{Q}^{-1} = \begin{pmatrix} \boldsymbol{Q}^{11} & \boldsymbol{Q}^{12} \\ \boldsymbol{Q}^{21} & \boldsymbol{Q}^{22} \end{pmatrix}, \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \text{and } \boldsymbol{b} = \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \end{pmatrix} \tag{22}$$

The conditional mean $\boldsymbol{\mu}_{1|2} = \mathrm{E}(\boldsymbol{x}_1 \mid \boldsymbol{x}_2)$ and precision matrix $\boldsymbol{Q}_{1|2} = (\mathrm{Var}(\boldsymbol{x}_1 \mid \boldsymbol{x}_2))^{-1}$ are given by

$$\begin{aligned} \boldsymbol{\mu}_{1|2} &= \mathrm{E}(\boldsymbol{x}_1) - \boldsymbol{Q}_{11}^{-1}\boldsymbol{Q}_{12}(\boldsymbol{x}_2 - \mathrm{E}(\boldsymbol{x}_2)) & (23) \\ \boldsymbol{Q}_{1|2} &= \boldsymbol{Q}_{11} + \mathrm{diag}(\boldsymbol{c}_1), & (24) \end{aligned}$$

where $\mathrm{E}(\boldsymbol{x}_1)$ and $\mathrm{E}(\boldsymbol{x}_2)$ are the marginal means, given by

$$\begin{pmatrix} \mathrm{E}(\boldsymbol{x}_1) \\ \mathrm{E}(\boldsymbol{x}_2) \end{pmatrix} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix} + \begin{pmatrix} \boldsymbol{Q}^{11} & \boldsymbol{Q}^{12} \\ \boldsymbol{Q}^{21} & \boldsymbol{Q}^{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \end{pmatrix}. \tag{25}$$

To sample from the conditional distribution, we need to compute the conditional mean and precision matrix. The conditional precision matrix is simply $\boldsymbol{Q}_{11}$, a sub-matrix of the full precision matrix. Thus, it's bandwidth will be equal to or less than for the full matrix. The conditional mean $\boldsymbol{\mu}_{1|2}$ is computed by solving

$$(\boldsymbol{Q}_{11} + \mathrm{diag}(\boldsymbol{c}_1))\, \boldsymbol{\mu}_{1|2} = \boldsymbol{b}^{'}, \tag{26}$$

were $\boldsymbol{b}^{'}$ is obtained by writing the Gaussian distribution of $\pi(\boldsymbol{x}_1 \mid \boldsymbol{x}_2)$ on the form

$$\pi(\boldsymbol{x}_1 \mid \boldsymbol{x}_2) \propto \exp\left(-\frac{1}{2}\boldsymbol{x}_1^T(\boldsymbol{Q}_{11} + \mathrm{diag}(\boldsymbol{c}_1))\, \boldsymbol{x}_1 + (\boldsymbol{b}^{'})^T \boldsymbol{x}_1\right). \tag{27}$$

Given the band-structure of $\boldsymbol{Q}_{11}$, this can be done efficiently, as described in Section 3.2.1.

3.4 Evaluation of the log-density of a sample

The routine `GMRFLib_evaluate` evaluates the log-density of a sample from a GMRF. The routine returns the full log-density, including the normalizing constant. For an unconditional sample $\boldsymbol{x}$, the log-density is computed as

$$\log\pi(\boldsymbol{x}) = -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{Q} + \mathrm{diag}(\boldsymbol{c})| - \frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T(\boldsymbol{Q} + \mathrm{diag}(\boldsymbol{c}))(\boldsymbol{x} - \boldsymbol{\mu}) + \boldsymbol{b}^T\boldsymbol{x}. \tag{28}$$

For a sample form the conditional distribution, the log-density is computed as the right hand side of

$$\pi(\boldsymbol{x}|\boldsymbol{A}\boldsymbol{x}) = \frac{\pi(\boldsymbol{A}\boldsymbol{x}|\boldsymbol{x})\pi(\boldsymbol{x})}{\pi(\boldsymbol{A}\boldsymbol{x})} \tag{29}$$

for deterministic linear constraints, and similarly for stochastic linear constraints. The routine `GMRFLib_evaluate` is called from within the sampling routine `GMRFLib_sample`, reusing terms computed during the sampling, to increase the efficiency. The routine can also be called by the user to evaluate the density of an arbitrary $\boldsymbol{x}$.

3.5 Block-sampling

In many applications, sampling from a GMRF $\boldsymbol{x}$ enters into Markov Chain Monte Carlo (MCMC) algorithms designed for sampling based estimation of the parameters of a model, given a set of data $\boldsymbol{y}$. Let ψ be the set of parameters to be estimated. In a general MCMC-algorithm, we generate approximate samples from the posterior distribution $\pi(\boldsymbol{\psi}|\boldsymbol{y})$, by sampling an ergodic Markov Chain having $\pi(\boldsymbol{\psi}|\boldsymbol{y})$ as it's stationary distribution. For an introduction to MCMC methods, see e.g. Robert and Casella, 1999 and Gilks et al., 1996.

A frequently used method is the Gibbs sampler, where at each iteration, each parameter is updated in turn, conditionally on the last updates of the remaining parameters. One or more of the parameter updates of the Gibbs sampler might involve a Metropolis-Hastings step. This routine works fine in many applications, but if the parameters to be estimated are highly correlated, the convergence and mixing might be very slow. A modification of the Gibbs sampler is to group the parameters in subsets or blocks, updating each block in turn.

One situation where this blocking approach is useful, is in the case where the parameters of the GMRF $\boldsymbol{x}$ depend on unknown hyper-parameters, that are to be estimated. It often turns out that several of these parameters are highly correlated to the GMRF $\boldsymbol{x}$, and thus the GMRF and the hyper-parameters should be updated jointly in one block. In `GMRFLib` a routine, `GMRFLib_blockupdate` is implemented to solve this task.

In general, the `GMRFLib_blockupdate` routine generates samples from a Gaussian Markov random field, or Gaussian proposals for a nearly Gaussian Markov random field, conditionally on a set of hyper-parameters $\boldsymbol{\theta}$. The most general model for the conditional density of $\boldsymbol{x}$ given $\boldsymbol{\theta}$ is

$$\pi(\boldsymbol{x}|\boldsymbol{\theta}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}(\boldsymbol{\theta}))^T(\boldsymbol{Q}(\boldsymbol{\theta})+\operatorname{diag}(\boldsymbol{c}(\boldsymbol{\theta})))(\boldsymbol{x}-\boldsymbol{\mu}(\boldsymbol{\theta}))+\boldsymbol{b}(\boldsymbol{\theta})^T\boldsymbol{x}+\sum d_i(\boldsymbol{\theta})f(x_i;\boldsymbol{\theta})\right). \quad (30)$$

If the functions $f(x_i;\boldsymbol{\theta})$ are linear or quadratic in $x_i;\ i=0,\ldots,n-1$, the model is Gaussian, but the functions can be general, and this corresponds to the cases we termed as nearly Gaussian above. The joint density of $\boldsymbol{x}$ and $\boldsymbol{\theta}$, from which we want to sample, is given by

$$\pi(\boldsymbol{x},\boldsymbol{\theta}) = \pi(\boldsymbol{x}|\boldsymbol{\theta})\pi(\boldsymbol{\theta}), \quad (31)$$

where $\pi(\boldsymbol{\theta})$ is the prior distribution of the hyper-parameters $\boldsymbol{\theta}$, and $\pi(\boldsymbol{x},\boldsymbol{\theta})$ is given by (30).

In the case of a MCMC update conditionally on data, the updates are to be computed from the joint posterior distribution of $\boldsymbol{x}$ and $\boldsymbol{\theta}$ given the data $\boldsymbol{y}$. In the posterior distribution

$$\pi(\boldsymbol{x},\boldsymbol{\theta}|\boldsymbol{y}) \propto \pi(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})\pi(\boldsymbol{x}|\boldsymbol{\theta})\pi(\boldsymbol{\theta}) \quad (32)$$

, the term $\pi(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})\pi(\boldsymbol{x}|\boldsymbol{\theta})$ should then be of the form (30), where the parameters and the functions $f(x_i)$ will depend on the data $\boldsymbol{y}$ in addition to the hyper-parameters $\boldsymbol{\theta}$. In such a case the functions $f(x_i;\boldsymbol{\theta});\ i=0\ldots,\mathtt{n}-1$ can, but do not have to, be equal to the elements of the log-likelihood of $\boldsymbol{x}$, given the data. An example of an application where this is the case, is described in detail in Section 5.4.

In a similar way as for the GMRF sampling methods described in Section 3.2, the block-sampling can be done conditionally on fixed values or a deterministic or stochastic linear constraint. The block-sampling algorithm is described in more detail in Section 4.4.

For an application of the block-sampling algorithm, see Knorr-Held and Rue, 2002.

4 Description and documentation of the C-routines

The routines implement operations on a set of basic objects, implemented as `structs` in the C routine library. These includes the graph specification `GMRFLib_graph_tp` and a set of `structs` specifying the setup for the simulation problems. These are all declared in the appropriate subsection.

In the documentations, pointer arguments to functions are marked (`input`) or (`output`), to emphasize which arguments will be altered by the routines. All array indices are running from 0 to $n-1$, where n is the length of the array. Also, node numbers are supposed to run from 0. Most functions return an `integer`. If not otherwise stated, this is an error code, being zero if the routine is run successfully. The error handling system is documented in more detail in Section 4.5.

4.1 Creating graphs and lattices

4.1.1 Storage scheme

In `GMRFLib`, a graph, whether a lattice or a general graph, is represented by the `struct GMRFLib_graph_tp`, declared by

```
typedef struct
{
    int n;                      /* number of nodes */
    int bandwidth;              /* bandwidth */
    int *nnbs;                  /* number of neighbors for each node */
    int **nbs;                  /* for each node: node numbers for neighbors */
    int *remap;                 /* reordering of nodes to reduce bandwith */
    int *mothergraph_idx;       /* indices for mother graph nodes (if subgraph) */

    GMRFLib_IsNeighb_tp *is_neighb; /* struct for checking for neighbors */
}
GMRFLib_graph_tp;
```

The members of a `GMRFLib_graph_tp`-object are defined in Table 2.

`n`:	The number of nodes in the graph.
`bandwidth`:	The bandwidth of the precision matrix $\boldsymbol{Q}$ for a GMRF $\boldsymbol{x}$ on the graph.
`nnbs`:	A length n array, where `nnbs[i]` contains the number of neighbours of node i; $i = 0; \ldots, n-1$.
`nbs`:	A length `n` array of arrays, where `nbs[i][j]` contains the node numbers j; $j = 0, \ldots, \text{nnbs}[j]-1$ for the neighbours of node i.
`remap`:	A re-ordering of the nodes to minimize the bandwidth. After re-ordering node number i in the real world appears in position `remap[i]`.
`mothergraph_idx`:	If the graph is a subgraph, `mothergraph_idx[i]` is the index of node i in the mothergraph of the current graph.
`is_neighb`:	An object for internal book-keeping of neighbours.

Table 2: Overview of the members of `GMRFLib_graph_tp`

This part of the code from `GMRFLib_print_graph` shows how to list all neighbours,

```
    fprintf(fp, "graph has %1d nodes and bandwidth %1d\n",
```

```
        graph->n, graph->bandwidth);
for(i=0;i<graph->n;i++)
{
    fprintf(fp, "node %1d has %1d neighbors:", i, graph->nnbs[i]);
    for(j=0;j<graph->nnbs[i];j++)
        fprintf(fp, " %1d", graph->nbs[i][j]);
    fprintf(fp, "\n");
}
```

There are two ways of specifying a general graph.

- *Reading from a file*. The simplest way to specify a general graph is to read the graph specifications from a file, using the routine `GMRFLib_read_graph`. This routine allocates and initializes the members based on the information given on the input file, and customizes the graph for use in other library functions.
- *Explicitly creating a* `GMRFLib_graph_tp` *-object*. The user can create a general graph by allocating and initializing a variable of type `GMRFLib_graph_tp`, and then (IMPORTANT!) calling the function `GMRFLib_prepare_graph` to customize the graph and check that the graph is consistently defined. The user should specify the the members `n`, `nnbs` and `nbs` in Table 2. The members `bandwidth`, `remap` and `is_neighb` are then allocated and initialized by calling the function `GMRFLib_prepare_graph`. The member `mothergraph_idx` is initialized, and needed, only if the graph is generated as a subgraph of another graph, using `GMRFLib_compute_subgraph`.

In addition to the routines for specifying general graphs, the library provides two separate functions for the specification of two-dimensional lattice graphs and one-dimensional linear graphs, and a separate setup for the generation of graphs for a weighted average model.

- *A lattice graph*: A two-dimensional graph on a lattice is most easily specified using the routine `GMRFLib_create_lattice`, specifying the grid sizes n_{row} and n_{col} and the parameters m_{row} and m_{col}, defining the neighbourhood $(2m_{\text{row}} + 1) \times (2m_{\text{col}} + 1)$.
- *A linear graph*: To specify a one-dimensional linear graph, i.e. an autoregressive AR(p)-model, use the routine `GMRFLib_make_linear_graph`, specifying the number of nodes and the size of the neighbourhood.
- *The wa-graph*: The neighbourhood-structure of the graph of on which a GMRF $\boldsymbol{x}$ is defined, is specified indirectly by specifying the weights, w_{ij}, and the graph corresponding to these weights, and then `GMRFLib` will compute the graph of $\boldsymbol{x}$. The graph is specified in three steps:
  1. Create a function `wafunc`, say, returning the weights $w_{ij};\ i = 1, \ldots, n;\ j = 1, \ldots, n$ for each pair of nodes i and j. The function should be of the same format as the function computing the elements of $\boldsymbol{Q}$ of a general graph, that is, of type `GMRFLib_Qfunc_tp`.
  2. Specify the graph of the weights w_{ij}, creating a `GMRFLib_graph_tp`-object.
  3. Call the library routine `GMRFLib_init_wa_problem`, creating the graph and the Q-function of the GMRF $\boldsymbol{x}$ corresponding to the user-defined weight function. This routine returns an object of type `GMRFLib_wa_problem_tp`, holding the graph and the Q-function needed for sampling and evaluation. The data structure is declared by

     ```
     typedef struct
     {
         GMRFLib_graph_tp *graph; /* graph on the GMRF x */
         GMRFLib_Qfunc_tp *Qfunc; /* the Q-function, computing Q(i,j) */
         char *Qfunc_arg;         /* the arguments to the Q-function */
     } GMRFLib_wa_problem_tp;
     ```

The members of the data structure can then be used just as the graph and the Q-function and it's arguments for a general graph.

The separate routines for creating and handling graphs are documented in the sub-sections that follow.

4.1.2 Routines for creating and handling general and regular graphs

GMRFLib_create_graph

Description: Creates an empty graph.

Declaration:

```
int GMRFLib_create_graph(GMRFLib_graph_tp **graph)
```

Arguments:

`graph (input, output):` A pointer to a `GMRFLib_graph_tp` pointer. At output, `graph` points to an empty graph.

Example:

```
main(int argc, char* argv[])
{
    /* ... */
    GMRFLib_graph_tp *graph;
    GMRFLib_create_graph(&graph);
    /* ... */
}
```

GMRFLib_free_graph

Description: Frees the memory held by a `GMRFLib_graph_tp`-variable. NOTE: To ensure safe memory handling, this function should only be applied to graphs generated by the graph-generating routines of the library, and NOT to user-specified graphs.

Declaration:

```
int GMRFLib_free_graph(GMRFLib_graph_tp *graph)
```

Arguments:

`graph (input,output):` A graph generated by a graph-generating library routine. At output, the graph and it's array members are all deallocated.

See also: `GMRFLib_read_graph`, `GMRFLib_make_linear_graph`, `GMRFLib_create_lattice`, `GMRFLib_prepare_graph`

GMRFLib_make_linear_graph

Description: Creates a linear graph, corresponding to an autoregressive $AR(p)$-model.

Declaration:

```
int GMRFLib_make_linear_graph(GMRFLib_graph_tp **graph, int n, int bw,
                              int cyclic_flag)
```

Argument: `graph (output)`: At output, (`*graph`) contains the specification of the linear graph, customized for use in other library routines.

`n`: The number of nodes of the graph.

`bw`: Specifies the bandwidth of the graph, such that each node has `bw` neighbours in each direction. A bandwidth of `bw` corresponds to an AR(`bw`)-model.

`cyclic_flag`: If this argument is > 0, the graph is made cyclic.

Example: See Section 5.1.

GMRFLib_read_graph

Description: Reads a graph from a file.

Declaration:

```
int GMRFLib_read_graph(GMRFLib_graph_tp **graph, char *filename)
```

Arguments:

`graph (input, output)`: At output, (`*graph`) has been initialized by using the information on the file `filename`.

`filename (input)`: the name of the file, formatted as described below, containing the specification of the graph.

Details: The file is assumed to be of the format

```
  n
nn[0]     nnbs[0]    nbs[nn[0]][0]     nbs[nn[0]][1]    ···   nbs[nn[0]][nnbs[0]-1]
nn[1]     nnbs[1]    nbs[nn[1]][0]     nbs[nn[1]][1]    ···   nbs[nn[1]][nnbs[1]-1]
  ⋮          ⋮             ⋮                 ⋮           ⋱             ⋮
nn[n-1]  nnbs[n-1]  nbs[nn[n-1]][0]  nbs[nn[n-1]][1]   ···  nbs[nn[n-1]][nnbs[n-1]-1]
```

Here, `n`, `nbs` and `nnbs` refer to the members of the struct `GMRFLib_graph_tp` described in Section 4.1.1, and `nn[i]` is the node number of the node with index i, running from 0 to $n-1$. In words, the first row contains the number of nodes, `n`, and the successive rows, rows $i = 2, \ldots, n+1$, list the node number, `nn[i]`, the number of neighbours, `nnbs[i]`, and the node numbers, `nbs[i][j]`, for the neighbours of each node i of the graph. The number of neighbours `nnbs[i]` might be zero.

The routine calls the routine `GMRFLib_prepare_graph` to customize the graph for further computations.

Example: See Section 5.1.

See also: `GMRFLib_prepare_graph`, `GMRFLib_print_graph`.

GMRFLib_print_graph

Description: Prints the specification of a graph to standard output or a file.

Declaration:

```
int GMRFLib_print_graph(FILE *fp, GMRFLib_graph_tp *graph)
```

Arguments:

- `fp (output)`: The `FILE*` on which to print the graph.
- `graph (input)`: The graph to be printed.

See also: `GMRFLib_read_graph`.

GMRFLib_validate_graph

Description: Validates the graph by checking that the members of a `GMRFLib_graph_tp`-object are defined consistently.

Declaration:

```
int GMRFLib_validate_graph(FILE *fp, GMRFLib_graph_tp *graph)
```

Arguments:

- `fp (output)`: If `!NULL`, the function will write error messages stating the type of validation error on this file, in addition to the default standard output error message issued by the `GMRFLib_error_handler`-function. If `NULL`, only the default error message is issued.
- `graph (input)`: The graph to be validated.

See also: `GMRFLib_error_handler`

GMRFLib_compute_subgraph

Description: Computes a subgraph of a graph by removing the nodes indicated by the `remove_flag` argument.

Declaration:

```
int GMRFLib_compute_subgraph(GMRFLib_graph_tp **subgraph,
                             GMRFLib_graph_tp *graph,
                             char *remove_flag)
```

Arguments:

- `subgraph (output)`: The subgraph generated by removing the nodes for which `remove_flag` is 1 from the graph `graph`.
- `graph (input)`: The graph from which to compute the subgraph.
- `remove_flag (input)`: An array of length n, the number of nodes in `graph`, specifying which nodes to be removed creating the subgraphs. The nodes for which `remove_flags` is 0, IS included in the subgraph.

Example: See Section 5.1.

GMRFLib_prune_graph

Description: Modifies a graph by removing entries where the elements of the precision matrix $Q(i,j) = 0, i \sim j$, and then recomputing the ordering of the nodes. NOTE: The new graph takes the same amount of memory as the old one and can be freed the same way (using `GMRFLib_free_graph`).

Declaration:

```
int GMRFLib_prune_graph(GMRFLib_graph_tp *graph,
                        GMRFLib_Qfunc_tp *Qfunc, char *Qfunc_arg)
```

Arguments:

`graph (input, output)`: A `GMRFLib_graph_tp`-object. At output, all pairs of elements of the graph corresponding to $Q(i,j) = 0$ are removed from the graph.

`Qfunc (input)`: A function defining the elements of the $\boldsymbol{Q}$-matrix.

`Qfunc_arg (input)`: A `character`-pointer defining the address of a variable or data structure holding the arguments to the function `Qfunc`.

See also: `GMRFLib_Qfunc_tp`

GMRFLib_prepare_graph

Description: Post-processes a user-specified `GMRFLib_graph_tp` object (see the introduction to Section 4.1.1), such that the resulting graph is of the format required by the library functions operating on graphs.

Declaration:

```
int GMRFLib_prepare_graph(GMRFLib_graph_tp *graph)
```

Arguments:

`graph (input, output)`: A graph explicitly specified by the user by creating and initializing a `GMRFLib_graph_tp`-object. At output, the graph is customized.

See also: `GMRFLib_read_graph`.

GMRFLib_Qx

Description: Computes $\boldsymbol{Qx}$, the matrix-vector product of the precision matrix $\boldsymbol{Q}$ and an array $\boldsymbol{x}$. The elements of $\boldsymbol{x}$ should be in the original ordering of the nodes of the graph.

Declaration:

```
int GMRFLib_Qx(double *result, double *x, GMRFLib_graph_tp *graph,
               GMRFLib_Qfunc_tp *Qfunc, char *Qfunc_arg)
```

Arguments:

`result (output)`: At output, the length `n` array `result` contains the value of the product $\boldsymbol{Qx}$.

`x (input)`: The array $\boldsymbol{x}$, as a length n array.

`graph (input)`: The graph on which $\boldsymbol{x}$ is defined.

Qfunc (input): The function defining the precision matrix $\boldsymbol{Q}$.

Qfunc_arg (input): A character-pointer defining the address of a variable or data structure holding the arguments to the function Qfunc.

See also: GMRFLib_Qfunc_tp

GMRFLib_fold_graph

Description: Returns a new graph created by expanding one graph g with another graph gg. The two graphs must have the same number of nodes.

Declaration:

```
int GMRFLib_fold_graph(GMRFLib_graph_tp **ng, GMRFLib_graph_tp *g,
                       GMRFLib_graph_tp *gg)
```

Arguments:

ng (output): The new expanded graph. The pointer (*ng) will be allocated within the routine.

g, gg (input): The graphs to be combined.

Details: Given two graphs g and gg, the new graph ng is created by expanding the neighbourhood of g by the neighbourhood of gg. That is, for each neighbour j_g of a node i_g of g, a node k_{gg} in gg is included in the neighbourhood of node i_g in g if k_{gg} and j_g are neighbours in gg. The new graph ng is equal to the expanded version of of g.

Example: See Section 5.1.

See also: GMRFLib_nfold_graph

GMRFLib_nfold_graph

Description: Returns a new graph created by multiple expanding of one graph by itself.

Declaration:

```
int GMRFLib_nfold_graph(GMRFLib_graph_tp **ng, GMRFLib_graph_tp *og,
                        int nfold)
```

Arguments:

ng (output): The new expanded graph. The pointer (*ng) will be allocated within the routine.

og (input): The graph to be expanded.

nfold: The number of times to expand the graph (see Details below).

Details: Given the old graph og, the new graph ng is created by expanding the neighbours of og by it's own neighbourhood nfold $-$ 1 times. The routine GMRFLib_fold_graph is called successively nfold-1 times, the first time using g=gg=og, and in successive runs letting g=gprev and gg=og, where gprev is the resulting graph of the previous run. For completeness, nfold $= 0$ is allowed for, meaning that a graph with n independent nodes is returned as ng, where n is the number of nodes in the old graph og. If nfold $= 1$, the new graph is a copy of the old graph.

Example: See Section 5.1.

See also: GMRFLib_fold_graph

GMRFLib_is_neighb

Description: Checks whether two nodes i and j are neighbours.

Declaration:

```
int GMRFLib_is_neighb(int node, int nnode, GMRFLib_graph_tp *graph)
```

Arguments:

`node, nnode`: The nodes i and j to be checked.

`graph (input)`: The graph to be checked.

Return value: Returns 1 if the nodes `node` and `nnode` are neighbours, otherwise it returns 0.

Example:

```
int main(int argc,  char **argv)
{
    GMRFLib_graph_tp *graph;  /* Graph object */
    int i, j, isn;

    /* create a non-cyclic graph on a lattice */
    GMRFLib_create_lattice(&graph, 10, 10, 1, 1, 0);

    /* check for neighbors */
    for(i=0;i<graph->n;i++)
        for(j=0;j<graph->n;j++)
        {
            isn = GMRFLib_is_neighb(i, j, graph);
            printf("%d and %d: %d\n", i, j, isn);
        }

    return 0;
}
```

GMRFLib_create_lattice

Description: Creates a graph on a lattice, customizing the graph for use in other library functions.

Declaration:

```
int GMRFLib_create_lattice(GMRFLib_graph_tp **graph, int nrow, int ncol,
                           int nb_row,  int nb_col, int cyclic_flag)
```

Arguments:

`graph (output)`: At output, (`*graph`) contains the specification of the lattice graph, customized for use in other library routines.

`nrow, ncol`: The number of rows (`nrow`) and columns (`ncol`) in the $n_{\text{row}} \times n_{col}$ lattice.

`nb_row, nb_col`: Specification of the parameters nb_{row} and nb_{col} of the $(2nb_{row} + 1) \times (2nb_{col} + 1)$-neighbourhood of the lattice.

`cyclic_flag`: If this argument is > 0, the graph is made cyclic.

Example: See Section 5.1.

See also: `GMRFLib_lattice2node`, `GMRFLib_node2lattice`

GMRFLib_lattice2node

Description: Defines the ordering of the nodes from the lattice. In the case of a rectangular neighbourhood, the ordering of the nodes will be the one minimizing the bandwidth of the precision matrix of a GMRF on the graph.

Declaration:

```
int GMRFLib_lattice2node(int *node, int irow, int icol, int nrow, int ncol)
```

Arguments:

`node (output)`: The node index ($\in [0, n_{\text{row}} \times n_{\text{col}} - 1]$) corresponding to the row and column indices of the graph.

`irow`: Row index ($\in [0, n_{\text{row}} - 1]$)

`icol`: Column index ($\in [0, n_{\text{col}} - 1]$)

`nrow`: The number of rows of the lattice

`ncol`: The number of columns of the lattice

Details: The node index `node` is computed so as to minimize the bandwidth of the precision matrix $\boldsymbol{Q}$ of a GMRF on the graph in the case of a rectangular neighbourhood, and will depend on the relative sizes of `nrow` and `ncol`:

$$\begin{array}{ll} \text{If } (\texttt{nrow} > \texttt{ncol}), & \texttt{node} = \texttt{irow} + \texttt{icol} * \texttt{nrow} \\ \text{If } (\texttt{nrow} \leq \texttt{ncol}) & \texttt{node} = \texttt{icol} + \texttt{irow} * \texttt{ncol} \end{array}$$

Example: See Section 5.1.

See also: `GMRFLib_node2lattice`

GMRFLib_node2lattice

Description: Extracts the lattice indices of the nodes of a lattice represented as a general graph.

Declaration:

```
int GMRFLib_node2lattice(int node, int *irow, int *icol, int nrow,
                         int ncol)
```

Arguments:

`node`: The node index ($\in [0, n_{\text{row}} \times n_{col} - 1]$) in the general graph representation.

`irow (output)`: row index ($\in [0, n_{\text{row}} - 1]$) corresponding to node number `node` in the graph.

`icol (output)`: Column index ($\in [0, n_{\text{col}} - 1]$) corresponding to node number `node` in the graph.

`nrow`: The number of rows of the lattice.

`ncol`: The number of columns of the lattice.

Details: The lattice indices `irow` and `icol` are computed so as to minimize the bandwidth of the precision matrix $\boldsymbol{Q}$ of a GMRF on the graph in the case of a rectangular neighbourhood. They will depend on the relative sizes of `nrow` and `ncol`:

$$\text{If } (\texttt{nrow} > \texttt{ncol}) \quad \begin{cases} \texttt{irow} = \texttt{node} - \texttt{icol} * \texttt{nrow} \\ \texttt{icol} = \texttt{node}/\texttt{nrow} \end{cases}$$

$$\text{If } (\texttt{nrow} \leq \texttt{ncol}) \quad \begin{cases} \texttt{irow} = \texttt{node}/\texttt{ncol} \\ \texttt{icol} = \texttt{node} - \texttt{irow} * \texttt{ncol} \end{cases}$$

Example: See Section 5.1.

See also: `GMRFLib_lattice2node`

GMRFLib_init_wa_problem

Description: This function converts a graph definition on the weights w_{ij} of a weighted average GMRF model, as defined by (8), to the graph defining the precision matrix $\boldsymbol{Q}$ of the GMRF $\boldsymbol{x}$.

Declaration:

```
int GMRFLib_init_wa_problem(GMRFLib_wa_problem_tp **wa_problem,
                            GMRFLib_graph_tp *wagraph,
                            GMRFLib_Qfunc_tp *wafunc,
                            char *wafunc_arg)
```

Arguments:

`wa_problem (output)`: At output, (`*wa_problem`) is allocated as a pointer to a `GMRFLib_wa_problem_tp`, holding the graph of the GMRF $\boldsymbol{x}$, the function defining the $\boldsymbol{Q}$-matrix and a `character`-pointer holding the address of the variable or data structure holding it's arguments.

`wa_graph (input)`: The graph of the weights $\{w_{ij}\}$ in the density function of $\boldsymbol{x}$, of the form (8).

`wafunc (input)`: A pointer to a function returning the weights w_{ij}. The function is to be of the same format as the function defining the elements of the $\boldsymbol{Q}$-matrix of a general GMRF, such that the argument list and return value should be the same as for the template `GMRFLib_Qfunc_tp`. The arguments should be the indices i and j and a `character`-pointer referring to additional arguments to the function. If $w_{ij} = 0$ initially, it is required to be kept unchanged.

`wafunc_arg (input)`: A `character`-pointer holding the address of a variable or data structure defining additional arguments to the function `wafunc`.

Details: Based on the weight function, defining the weights in (8), the routine computes the graph corresponding to the precision matrix $\boldsymbol{Q}$ of the GMRF x. Also, the function defining the elements of $\boldsymbol{Q}$ and the set of arguments (in addition to the indices i and j) are generated. These are returned as members of the `GMRFLib_wa_problem_tp`-object `(*problem)`, this data structure is declared in Section 4.1.1.

NOTE: The weights w_{ij} that are initially defined to be 0, should be kept fixed. The routine `GMRFLib_prune_graph` is called on the resulting graph of the GMRF $\boldsymbol{x}$, removing elements of the graph corresponding to $Q(i,j) = 0$. Re-setting the zero weights, for example by letting the weights depend on parameters that might change while running the programs, will invalidate this graph reduction.

Example: See Section 5.3.

See also: `GMRFLib_free_wa_problem`

GMRFLib_free_wa_problem

Description: Deallocates a `GMRFLib_wa_problem_tp`-object and it's members, allocated by the routine `GMRFLib_init_wa_problem`.

Declaration:

```
int GMRFLib_free_wa_problem(GMRFLib_wa_problem_tp *wa_problem)
```

Arguments:

`wa_problem (intput, output)`: A `GMRFLib_wa_problem_tp`-object, allocated by the routine `GMRFLib_init_wa_problem`. At output, `wa_problem` and it's members have been deallocated.

Example: See Section 5.3.

See also: `GMRFLib_init_wa_problem`

4.2 Definition of the precision matrix Q

GMRFLib_Qfunc_tp

Description: A template for functions defining the precision matrix $\boldsymbol{Q}$ of a GMRF $\boldsymbol{x}$.

Declaration:

```
typedef double GMRFLib_Qfunc_tp(int node, int nnode, char *argument)
```

Arguments:

`node, nnode`: The nodes of the graph for which to compute the precision. **Note that node and nnode are always to be neighbours, or equal!**

`argument`: A `character`-pointer holding the address of a variable or data structure defining the arguments to the function.

Return value: The value $Q(node, nnode)$, that is, the element of the precision matrix corresponding to nodes `node` and `nnode`.

Details: The precision matrix definition is problem specific, and should be provided by the user by implementing a function of the stated format. A pointer to a `GMRFLib_Qfunc_tp` is needed as an argument to the routine `GMRFLib_init_problem`, initializing the sampling problem.

Example: Consider the precision matrix

$$Q_{ij} = \begin{cases} \texttt{nnbs}[i] & \text{if } i == j \\ -1 & \text{if } i \neq j, \end{cases}$$

where $\texttt{nnbs}[i]$ is the number of neighbours of node i in the graph. This is the precision matrix $\boldsymbol{Q}$ of a GMRF with density of the form

$$\pi(\boldsymbol{x}) \propto \exp(-\frac{1}{2}\sum_{i\sim j}(x_i - x_j)^2) = \exp(-\frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x}) \qquad (35)$$

A function of type `GMRFLib_Qfunc_tp` implementing this precision matrix is

```
double Q_func(int node, int nnode, char *arg)
{
    GMRFLib_graph_tp *graph = (GMRFLib_graph_tp *) arg;

    if (node != nnode)
        return -1.0;
    else
        return graph->nnbs[node];
}
```

In this case, the problem specific `char*` argument is a `graph`-pointer, and the function uses a cast from `char*` to `graph*` to access the `graph`-object. Note that the function requires nodes `node` and `nnode` to be neighbours or equal!

See also: `GMRFLib_Qx`

4.3 Sampling from a GMRF

Sampling,whether unconditionally or conditionally, from a GMRF on a general graph, is performed using one function, `GMRFLib_sample`. Similarly, the function `GMRFLib_evaluate` computes the log-density for a sample from a GMRF. The functions operate on a data structure, `GMRFLib_problem_tp`, holding all external and internal information needed by the sampling and evaluation algorithms. This data structure is initialized by the function `GMRFLib_init_problem`.

4.3.1 Storage schemes

Specification of the sampling problem

The simulation routines operate on objects of type `GMRFLib_problem_tp`, which is declared below. The members of the `struct` are all initialized and operated on within the library routines. Many of the members are internal, and only the members needed by the user for extracting the results of the sampling routines, are shown; these are described in Table 4.3.1. The members `sub_mean` and `sub_mean_constr` are of length `n_sub`, where `n_sub` is the number of elements of the GMRF $\boldsymbol{x}$ that are updated during the sampling. Which elements of $\boldsymbol{x}$ that are updated, is given by the index vector `map`. More specifically, the elements `map`$[i]$; $i = 0, \ldots,$ `n_sub` $- 1$ of $\boldsymbol{x}$ are updated, and these correspond to the elements $i = 0, \ldots,$ `n_sub` $- 1$ of the mean vectors `sub_mean` and `sub_mean_constr`. Examples are given in Section 5.2.

```
typedef struct
{
    double *sample;                   /* the sample (full graph) */

    int *map;                         /* mapping to the real-world */

    double sub_logdens;               /* the log-density of sub_sample */
    double *sub_mean;                 /* mean (subgraph) without constraint */
    double *sub_mean_constr;          /* mean (subgraph) with constraint */

    GMRFLib_graph_tp     *sub_graph;   /* the subgraph */
    /* ... */
}
GMRFLib_problem_tp;
```

sample	A length n array containing a sample of the GMRF. If some of the elements of $\boldsymbol{x}$ are fixed, these fixed values are copied into the corresponding positions in sample, while the rest are samples from the conditional distribution given the fixed values.
sub_logdens	The log-density of the sample sample, corresponding to the elements of $\boldsymbol{x}$ that are not fixed.
map	The mapping between the elements of sample and sub_mean, sub_mean_constr and sub_graph. The correspondence is defined by sub_mean[i] $\leftrightarrow$ sample[map[i]].
sub_mean	The conditional mean of the sampled elements of $\boldsymbol{x}$, given the values of the fixed elements. These are the mean values of an *unconstrained* sample, that is, before taking into account any constraints.
sub_mean_constr	The mean of the variables corresponding to the sampled values in sample. This is the conditional mean of the sampled elements of $\boldsymbol{x}$, given the values of the fixed elements and *after* taking any constraints into account.
sub_graph	The subgraph defined on the nodes for which the elements of $\boldsymbol{x}$ are not fixed. This can be used to extract sub_graph$\rightarrow$n, the number of elements of the graph, equal to the length of the map-array

Table 3: The external members of GMRF_problem_tp

Specification of constraints

For conditional simulation on a linear deterministic or stochastic constraint, the constraint, of the form $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e} + \boldsymbol{\epsilon}$, is defined by specifying the members of the data structure GMRFLib_constr_tp, declared by

```
typedef struct
{
    int nc;                    /* number of constaints */
    double *a_matrix;          /* A matrix */
    double *e_vector;          /* e vector */
    double *errcov_diagonal;   /* if non-NULL, a diagonal error cov-matrix */
    double *errcov_general;    /* if non-NULL, a general error cov-matrix */

    GMRFLib_constr_intern_tp *intern;  /* for internal use mainly */
}
GMRFLib_constr_tp;
```

The type GMRFLib_constr_intern_tp is declared by

```
typedef struct
{
    double *chol;   /* = chol(errcov_*) */
    double *logdet; /* = log(det(errcov_*)) */
    char *Qpattern; /* = the zero/non-zero pattern of inv(errcov_*) */
}
GMRFLib_constr_intern_tp;
```

This last data structure is mainly intended for internal usage, holding the Cholesky decomposition (chol) and the log of the determinant (logdet) of the covariance matrix $\boldsymbol{\Sigma}_\epsilon$.

In the case of a deterministic constraint, the last three members of GMRFLib_constr_tp are not needed, and the members errcov_diagonal and errcov_general should be set to NULL. If one or both of these members are !NULL, the constraint is assumed to be stochastic, and the last three members specify the covariance matrix $\boldsymbol{\Sigma}_\epsilon$ of $\boldsymbol{\epsilon}$ and it's decomposition. The matrix $\boldsymbol{\Sigma}_\epsilon$ is required to be non-singular. The members of the `struct` are defined in Table 4.3.1.

For a deterministic or stochastic constraint:	
`nc`	The number of constraints, can be 0.
`a_matrix`	A length (`nc` $\times$ `n`) array holding the elements of the `nc` $\times$ `n` matrix $\boldsymbol{A}$ in the linear constraint $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e}$. The matrix is assumed to be stored column by column, that is, element $i + nc * j$ of `a_matrix` should correspond to $A[i, j]$.
`e_vector`	A length `nc` array holding the elements of the vector $\boldsymbol{e}$ of the constraint $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e}$.
For a stochastic constraint:	
`errcov_diagonal`	If `!=NULL`, the covariance matrix $\boldsymbol{\Sigma}_\epsilon$ of the error $\boldsymbol{\epsilon}$ is assumed to be diagonal, and `errcov_diagonal` is a length `nc`-array containing the diagonal elements.
`errcov_general`	If `!=NULL`, `errcov_general` specifies a general positive definite covariance matrix $\boldsymbol{\Sigma}_\epsilon$ of the error $\boldsymbol{\epsilon}$. If both `errcov_general` and `errcov_diagonal` are `!NULL`, the matrix is assumed to be diagonal, and `errcov_diagonal` is used.

Table 4: The members of the `GMRFLib_constr_tp` data structure.

To simplify allocation, initialization and deallocation of a `GMRFLib_constr_tp`-object, a set of functions operating on this data structure is implemented. To ensure safe memory handling, the user is recommended to use these function, which are documented in the next subsection. A `GMRFLib_constr_tp`-object is created and initialized by the following steps:

1. Allocate memory by calling `GMRFLib_create_constr`.
2. Set the number of constraints `nc`, and explicitly allocate the arrays `a_matrix` and `e_vector` by using `calloc`, and fill in the values of their elements. The number of elements of the GMRF (the number of columns of $\boldsymbol{A}$), will be defined within the corresponding `GMRFLib_graph_tp`-object, for which the constraints are defined.
3. Compute internal information, customizing the object for use in the sampling routines, by calling `GMRFLib_prepare_constr`. If the constraint is deterministic, nothing is to be done, and the function is empty.

Two other operations on constraints are available:

- To print the contents of the `GMRFLib_constr_tp`-object, call `GMRFLib_print_constr`.
- To evaluate the expressions $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{e}$ and $(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{e})^T \boldsymbol{Q}(\boldsymbol{A}(\boldsymbol{x} - \boldsymbol{e})$ for a given value of the GMRF $\boldsymbol{x}$, call the routine `GMRFLib_eval_constr`.

4.3.2 Routines for creating and handling constraints

GMRFLib_create_constr

Description: Creates a pointer to a GMRFLib_constr_tp-object.

Declaration:

```
int GMRFLib_create_constr(GMRFLib_constr_tp **constr);
```

Arguments:

constr (intput, output): A pointer to a GMRFLib_constr_tp pointer. At output, constr points to a GMRFLib_constr_tp-object. The internal pointer member intern points to an empty GMRFLib_constr_intern_tp-object, the other members are uninitialized.

Details: This functions allocates the memory needed by the GMRFLib_constr_tp-object and the GMRFLib_constr_intern_tp-object. To initialize the members of (*constr), allocate a_matrix and e_vector directly, using calloc, define nc and call GMRFLib_prepare_constr.

See also: GMRFLib_free_constr, GMRFLib_prepare_constr

GMRFLib_eval_constr

Description: Evaluates the expressions $\boldsymbol{Ax} - \boldsymbol{e}$ and $(\boldsymbol{Ax} - \boldsymbol{e})^T \boldsymbol{Q}(\boldsymbol{Ax} - \boldsymbol{e})$ for a given value of the GMRF $\boldsymbol{x}$.

Declaration:

```
int GMRFLib_eval_constr(double *value, double *sqr_value,
                        double *x,  GMRFLib_constr_tp *constr,
                        GMRFLib_graph_tp *graph)
```

Arguments:

value (output): A length constr->nc array holding the values of $\boldsymbol{Ax} - \boldsymbol{e}$. If NULL, $\boldsymbol{Ax} - \boldsymbol{e}$ is not evaluated.

sqr_value (output): A length constr->nc array holding the value of $(\boldsymbol{Ax} - \boldsymbol{e})^T \boldsymbol{Q}(\boldsymbol{Ax} - \boldsymbol{e})$. If NULL, $(\boldsymbol{Ax} - \boldsymbol{e})^T \boldsymbol{Q}(\boldsymbol{A}(\boldsymbol{x} - \boldsymbol{e})$ is not evaluated.

x (input): The instance of the GMRF, $\boldsymbol{x}$, for which to compute the constraints.

constr (input): A pointer to a GMRFLib_constr_tp-object holding the information on the type of linear constraint.

graph (input): The graph on which the GMRF and the constraint are defined.

GMRFLib_free_constr

Description: Deallocates the memory held by a GMRFLib_constr_tp-object. NOTE: To ensure safe memory handling, this function should only be applied to pointers allocated by using the library routines GMRFLib_create_constr and GMRFLib_prepare_constr. Also, as long as nc > 0, the arrays a_matrix and e_vector should both be !NULL, and should have been dynamically allocated (using calloc).

```
int GMRFLib_free_constr(GMRFLib_constr_tp *constr);
```

Arguments:

`constr (intput, output):` A pointer to a `GMRFLib_constr_tp`-object. At output, the pointer `constr` and it's member pointers are deallocated.

See also: `GMRFLib_create_constr`

GMRFLib_print_constr

Description: Prints the available information on a constraint held by an `GMRFLib_constr_tp`-object.

Declaration:

```
int GMRFLib_print_constr(FILE *fp, GMRFLib_constr_tp *constr,
                         GMRFLib_graph_tp *graph);
```

Arguments:

`fp:` The `FILE*` on which to print the constraints.

`constr (intput):` A pointer to a `GMRFLib_constr_tp`-object.

`graph (input):` The graph on which the constraints are defined.

See also: `GMRFLib_create_constr`

GMRFLib_prepare_constr

Description: Prepare a constraint by computing the information in the `intern` object, if needed.

Declaration:

```
int GMRFLib_prepare_constr(GMRFLib_constr_tp *constr);
```

Arguments:

`constr (intput, output):` At input, a pointer to a `GMRFLib_constr_tp`-object, created by `GMRFLib_constr_tp`. At output, internal information needed by the sampling routines is computed and added to the object.

Details: The function computes the Cholesky factorization and the log of the determinant of the covariance matrix $\boldsymbol{\Sigma}_\epsilon$. The matrix $\boldsymbol{A}$ and the vector $\boldsymbol{e}$ of the constraint $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e}$ should be allocated and initialized explicitly. This can be done before or after calling `GMRFLib_prepare_constr`.

See also: `GMRFLib_create_constr`, `GMRFLib_free_constr`

4.3.3 Sampling and evaluation routines

GMRFLib_init_problem

Description: Initializes and specifies a `GMRFLib_problem_tp`-object holding all information needed for sampling from a GMRF and evaluating the log-density of such a sample.

Declaration:

```
int GMRFLib_init_problem(GMRFLib_problem_tp **problem, double *x,
                         double *b, double *c, double *mean,
                         GMRFLib_graph_tp *graph,
                         GMRFLib_Qfunc_tp *Qfunc, char *Qfunc_args,
                         char *fixed_value,
                         GMRFLib_constr_tp *constraint,
                         unsigned int keep)
```

Arguments:

`problem (output, input)`: At output, `(*problem)` is a pointer to a `GMRFLib_problem_tp`-object, initialized and defined according to the problem specification. The required value of `(*problem)` at input depends on the argument `keep`. If `keep` $= 0$, `(*problem)` is allocated within the routine, and thus expected to be `NULL` at input. If `keep` > 0, `(*problem)` is assumed to be an already initialized `GMRFLib_problem_tp`-pointer at input. See also `keep` and `Details`.

`x (input)`: A length `n` array, where `n` is the number of nodes in the graph, of initial values of the GMRF. If `fixed_value` $\neq$ `NULL`, the elements of $\boldsymbol{x}$ corresponding to `fixed_value` $= 1$ are the fixed values in a conditional simulation. The remaining elements of $\boldsymbol{x}$ are not used, and can take arbitrary values. If `fixed_value` $=$ `NULL`, all values can be arbitrary, but $\boldsymbol{x}$ should not be `NULL`.

`b (input)`: If `!NULL`, a length `n` array holding the elements of the vector $\boldsymbol{b}$ in the general expression of the density if a GMRF, as given in (2).

`c (input)`: If `!NULL`, a length `n` array of elements to add to the diagonal of the precision matrix defined by the function `Qfunc`. The argument `c` should hold the elements of the vector $\boldsymbol{c}$ in (2).

`mean (input)`: If `!NULL`, a length `n` array holding the elements of the vector $\boldsymbol{\mu}$ in (2). If $\boldsymbol{b} = 0$, `mean` is the mean of the GMRF.

`graph (input)`: The graph on which the GMRF is defined.

`Qfunc (input)`: A pointer to a user-defined function defining the precision matrix $\boldsymbol{Q}$ of a GMRF $\boldsymbol{x}$.

`Qfunc_args (input)`: The arguments to the function `Qfunc` defining the precision matrix.

`fixed_value (input)`: If `!NULL`, the sampling is done conditionally on fixed values. The elements of this array, of length `n`, should take the value 0 or 1. The conditional mean and covariance matrix of $\{x_i : \texttt{fixed\_value}[i] = 0\}$ given $\{x_i : \texttt{fixed\_value}[i] = 1\}$ are computed, and by sampling using `GMRFLib_sample`, elements of the GMRF $\boldsymbol{x}$ corresponding to `fixed_value` $= 1$, are kept fixed at the corresponding values specified in the argument `x`.

`constraint (input)`: If `!NULL`, a pointer to a `GMRFLib_constr_tp`-variable holding information about a linear (deterministic or stochastic) constraint (see Section 4.3.1). If `NULL`, the sampling is unconstrained.

keep: If 0, (*problem) is allocated within the routine, and thus expected to be NULL at input. If > 0, the (*problem)-pointer is supposed to be already initialized, pointing at a problem specification object, of which parts of the problem specification is to be kept unchanged. Which parts to keep fixed, are defined by the actual value of keep. See Details below.

Details: This function is to be called prior to sampling and evaluation of the log-density. If keep $= 0$ (or GMRFLib_NEW_PROBLEM), the GMRFLib_problem_tp-pointer (*problem) is dynamically allocated within the function. If keep > 0, (*problem) is required to be non-NULL, and is not allocated within the routine. On or more of the members (*problem) is to be kept fixed and not be re-allocated. What parts of the problem specification to keep fixed during sampling, is specified by the value of keep. It should take one out of four predefined macro values, or combinations of these. The choices are

GMRFLib_KEEP_mean	Keep the computed conditional mean value
GMRFLib_KEEP_graph	Keep the specified graph
GMRFLib_KEEP_bchol	Keep the Cholesky factorization of the $\boldsymbol{Q}$-matrix.
GMRFLib_KEEP_constr	Keep the constraints

Two or more of the keep-values are combined by letting keep = val1|val2|... The allocated memory of a GMRFLib_problem_tp-pointer is freed using the function GMRFLib_free_problem. Letting fixed_value $\neq$ NULL, the problem is defined to be a conditional sampling problem, conditioning on elements for which fixed_value $= 1$. With constraint $\neq$ NULL, sampling will be done conditionally on a linear constraint. If both fixed_value and constraint are NULL, unconditional samples will be generated by GMRFLib_sample. The arguments problem, x, graph and Qfunc are all required to be !NULL at input.

Example: See Section 5.

See also: GMRFLib_Qfunc_tp, GMRFLib_sample, GMRFLib_evaluate, GMRFLib_free_problem.

GMRFLib_free_problem

Description: Deallocates all allocated arrays in a GMRFLib_problem_tp-object.

Declaration:

```
int GMRFLib_free_problem(GMRFLib_problem_tp *problem)
```

Arguments:

problem (input, output): At output, all dynamically allocated array members of problem are deallocated.

See also: GMRFLib_init_problem

GMRFLib_sample

Description: Samples one realization of the elements of a GMRF $\boldsymbol{x}$. Whether the sampling is constrained or unconstrained, or should be done conditionally on fixed values, is defined by the GMRFLib_problem_tp-argument.

Declaration:

```
int GMRFLib_sample(GMRFLib_problem_tp *problem)
```

Arguments:

`problem (input, output):` At input `problem` should contain the problem specification, as defined by a call to `GMRFLib_init_problem`. At output, a sample of the GMRF has been generated, and is stored in the `sample` member of the data structure `problem`. Also, the log-density of the sample is computed, and stored in the `sub_logdens` member. If `problem` = `NULL`, nothing is done, and the routine returns 0.

Details: To sample one realization from a GMRF $\boldsymbol{x}$, first invoke `GMRFLib_init_problem` generating the problem specification by initializing a `GMRFLib_problem_tp`- object `problem`, and then call `GMRFLib_sample` using `problem` as an argument. Repeated samples are generated by repeated calls to `GMRFLib_sample`, using the same problem specification object. There is no need to call `GMRFLib_init_problem` more than once for each sampling problem. The log-density of the sample is computed within the sampling routine, but can also be computed explicitly for a `GMRFLib_problem_tp`-object by calling the routine `GMRFLib_evaluate`.

Prior to sampling, the built-in random number generator has to be initialized. This is done by calling the function `(*GMRFLib_uniform_init)`, taking one integer argument, specifying the seed of the generator. The user might replace the default random generator by another, see Section 4.6 for details.

At output, the sampling results are stored in the `GMRFLib_problem_tp`-object. No functions are provided for extracting the results of the sampling, so the results are retrieved by explicitly accessing the members of `problem` after running the routine. The sampled values are stored in `problem` $\rightarrow$ `sample` and the corresponding log-density in `problem` $\rightarrow$ `sub_logdens`. The fixed values are copied into the corresponding elements in `sample`.

Example: See Section 5.

See also: `GMRFLib_init_problem`, `GMRFLib_evaluate`

GMRFLib_evaluate

Description: Evaluates the log-density of a sample.

Declaration:

```
int GMRFLib_evaluate(GMRFLib_problem_tp *problem)
```

Arguments:

`problem (input, output):` At input `problem` should contain the problem specification and a sample, as defined by a call to `GMRFLib_init_problem` followed by one or more calls to `GMRFLib_sample`. At output, the log-density of the current sample has been computed, and is stored in the member `log_subdens` of `problem`.

Details: To extract the value of the log-density, the user should access the member `sub_logdens` of `problem` directly.

Example: See Section 5.

See also: `GMRFLib_init_problem`, `GMRFLib_sample`

4.4 Block-sampling

The block-sampling algorithm implements Markov Chain Monte Carlo (MCMC)-updating of a GMRF $\boldsymbol{x}$ and it's hyper-parameters $\boldsymbol{\theta}$ jointly in one block. The joint distribution of $\boldsymbol{x}$ and $\boldsymbol{\theta}$, possibly conditioning on data, is supposed to be of the general form (30).

The block-update routine `GMRFLib_blockupdate` returns a sample $\boldsymbol{x}^{'}$ from a Gaussian random field, conditional on a sample $\boldsymbol{\theta}^{'}$ of the hyper-parameters $\boldsymbol{\theta}$. The sample might be conditional on fixed values or on a linear deterministic or stochastic constraint; what method to use is specified in the same way as for the `GMRFLib_sample`-routine, documented in Section 4.3.

The routine also computes the part of log acceptance probability corresponding to the update of $\boldsymbol{x}$. The acceptance probability of the step $(\boldsymbol{x}, \boldsymbol{\theta}) \rightarrow (\boldsymbol{x}^{'}, \boldsymbol{\theta}^{'})$, given the data $\boldsymbol{y}$, is given by $\min\{1, R\}$, where

$$
\begin{aligned}
R &= \frac{\pi(\boldsymbol{x}', \boldsymbol{\theta}^{'} \mid \boldsymbol{y})}{\pi(\boldsymbol{x}, \boldsymbol{\theta} \mid \boldsymbol{y})} \times \frac{q(\boldsymbol{x}^{'} \rightarrow \boldsymbol{x} \mid \boldsymbol{\theta})\, q(\boldsymbol{\theta}^{'} \rightarrow \boldsymbol{\theta})}{q(\boldsymbol{x} \rightarrow \boldsymbol{x}' \mid \boldsymbol{\theta}^{'})\, q(\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}^{'})} \\
&= \frac{\pi(\boldsymbol{y} \mid \boldsymbol{x}^{'}, \boldsymbol{\theta}^{'})\pi(\boldsymbol{x}^{'} \mid \boldsymbol{\theta}^{'})}{\pi(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{\theta})\pi(\boldsymbol{x} \mid \boldsymbol{\theta})} \frac{q(\boldsymbol{x}^{'} \rightarrow \boldsymbol{x} \mid \boldsymbol{\theta})}{q(\boldsymbol{x} \rightarrow \boldsymbol{x}' \mid \boldsymbol{\theta}^{'})} \times \frac{\pi(\boldsymbol{\theta}^{'})}{\pi(\boldsymbol{\theta})} \frac{q(\boldsymbol{\theta}^{'} \rightarrow \boldsymbol{\theta})}{q(\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}^{'})} \\
&= R_x \times R_\theta \qquad (36)
\end{aligned}
$$

The contribution computed by `GMRFLib_blockupdate` is $\log(R_x)$ EXCEPT FOR the log ratio of the normalizing constant of the prior $\pi(\boldsymbol{x} \mid \boldsymbol{\theta})$ of $\boldsymbol{x}$, evaluated in $\boldsymbol{\theta}^{'}$ and $\boldsymbol{\theta}$. The normalizing constant might depend on the unknown hyper-parameters to be updated, and thus has to be taken into account. The log-ratio of the normalizing constants, as well as the value $\log(R_\theta)$, which is the contribution from the update of the hyper-parameters, have to be added explicitly in the application program.

To summarize, a joint update of $(\boldsymbol{x}, \boldsymbol{\theta})$, with prior distribution $\pi(\boldsymbol{x} \mid \boldsymbol{\theta})\pi(\boldsymbol{\theta})$ is generated by running trough the following steps

1. Generate a proposal $\boldsymbol{\theta}^{'}$ from $q(\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}^{'})$.
2. Using the routine `GMRFLib_blockupdate`, generate a proposed value $\boldsymbol{x}^{'}$ from $q(\boldsymbol{x} \rightarrow \boldsymbol{x}^{'} | \boldsymbol{\theta}^{'}) = \tilde{\pi}(\boldsymbol{x} \mid \boldsymbol{\theta}^{'}, \boldsymbol{y})$, where $\tilde{\pi}$ is the Gaussian approximation to the posterior $\pi(\boldsymbol{x} \mid \boldsymbol{\theta}^{'}, \boldsymbol{y})$. From the routine, the contribution to the log acceptance rate from the unnormalized prior and the proposal distribution for $\boldsymbol{x}$ is also obtained.
3. Add the log-ratio of the normalization constants of the prior of the GMRF and the log ratios of the prior and proposal distribution of $\boldsymbol{\theta}$ to the log acceptance rate.
4. Accept or reject the joint proposal $(\boldsymbol{x}^{'}, \boldsymbol{\theta}^{'})$.

An example illustrating how to implement a program using the block-update routine, is given in Section 5.2.

In the next subsection we describe some data structures needed in the updating routine, and then we describe the block-update routine itself and a template for computing the log-likelihood of a sample from a GMRF $\boldsymbol{x}$ given data.

4.4.1 Storage schemes

Here we describe two data structures, holding information on how to set up the block-sampling algorithm and the optimizer used to compute the current mode of the posterior distribution.

Setting up the block-sampler

Within the block-sampling algorithm, a quadratic approximation to the posterior distribution (30) is computed such that the sampling algorithms for GMRF's declared in Section 4.3 can be used. The quadratic approximation is computed by a Taylor expansion around the mode of the posterior distribution, found by an internal optimizer, or around the current value of $\boldsymbol{x}$. Which method to choose is defined by the value of the data structure `GMRFLib_blockupdate_param_tp`. This data structure is also used to choose whether or not to display the calculated values of likelihoods and acceptance rates for the updates, and the name of the file on which to print the calculations. The data structure is declared by

```
typedef struct
{
    int modeoption;    /* mode or current value */
    FILE *fp;          /* to display calculations */
}
GMRFLib_blockupdate_param_tp;
```

The members of the data structure, together with their default values, are described in Table 5. The simplest way to create a `GMRFLib_blockupdate_param_tp`-object holding the default values, is by calling the function `GMRFLib_default_blockupdate_param`. This is declared by

```
int GMRFLib_default_blockupdate_param(GMRFLib_blockupdate_param_tp
                                      **blockupdate_par).
```

Variable	*Default value*	*Description*
`modeoption`	0	If `GMRFLib_MODEOPTION_MODE`, the mode of the posterior distribution is estimated and the expansion is done around the mode. If `GMRFLib_MODEOPTION_CURRENT`, the expansion is done around the current value of $\boldsymbol{x}$.
`fp`	`NULL`	If `!NULL`, the file on which to print calculated likelihood values and acceptance rates. If `NULL`, no values are printed.

Table 5: The members of the data structure `GMRFLib_blockupdate_param_tp` and their default values.

Specifying the optimizer

The data structure `GMRFLib_optimize_param_tp` is used to specify the options for the optimizer.

```
typedef struct
{
    FILE *fp;
    int opt_type;
    int nsearch_dir;
    int restart_interval;
    int max_iter;
    int max_linesearch_iter;
    double abserr_func;
    double abserr_step;
    double step_len;
```

```
}
GMRFLib_optimize_param_tp;
```

If no object of type `GMRFLib_optimize_param_tp` is specified and supplied to the block-updating-routine `GMRFLib_blockupdate`, default values are used. These are listed in Table 6, along with a description of the members of the data structure. Also, a function `GMRFLib_default_optimize_param`, declared by

```
int GMRFLib_default_optimize_param(GMRFLib_optimize_param_tp **optpar)
```

is provided, creating a `GMRFLib_optimize_param_tp`-object holding the default settings. To change the default settings, a `GMRFLib_optimize_param_tp`-object should be dynamically allocated and the members should be set to the preferred values. A pointer to this object should then be supplied as an argument to `GMRFLib_blockupdate`.

Variable	*Default value*	*Description*
`fp`	`NULL`	A file for printing output from the optimizer.
`opt_type`	`GMRFLib_OPTTYPE_SAFENR`	The type of optimizer to be used. Four methods are available, see Table 7.
`nsearch_dir`	1	Indicates the number of previously search gradient directions on which the current search direction should be orthogonal (using the conjugate gradient (CG) method).
`restart_interval`	10	If `restart_interval` $= r$, the CG search will be restarted every r'th iteration.
`max_iter`	50	The maximum number of iterations.
`max_linesearch_iter`	50	The maximum number of iterations within each search direction (CG).
`step_len`	1.0e-4	Step length in the computation of a Taylor expansion or second order approximation of the log-likelihood around a point $\boldsymbol{x}_0$ (CG and NR).
`abserr_func`	0.5e-3	The absolute error tolerance for the value of the function to be optimized (CG and NR).
`abserr_step`	0.5e-3	The absolute error tolerance, relative to the number of nodes, for the size of one step of the optimizer (CG and NR).

Table 6: The members of the data structure `GMRFLib_optimize_param_tp`, specifying the optimization options.

The optimizer will use one out of four methods, listed in Table 7. The first two are the ordinary conjugate gradient (CG) and Newton-Raphson (NR) methods, while the last two are expected to be safer, doing the optimization in two steps: First finding the optimum using a diagonal precision matrix $\boldsymbol{Q}$, an then adjusting for the correlations.

Type specification	*Description*
`GMRFLib_OPTTYPE_CG`	The conjugate gradient method
`GMRFLib_OPTTYPE_NR`	The Newton-Raphson method
`GMRFLib_OPTTYPE_SAFECG`	A two-step conjugate gradient method
`GMRFLib_OPTTYPE_SAFENR`	A two-step Newton-Raphson method

Table 7: The four optimizing methods available

4.4.2 The block-updating-routine

GMRFLib_blockupdate

Description: This routine

Declaration:

```
int GMRFLib_blockupdate(double *laccept,
                        double *x_new, double *x_old,
                        double *b_new, double *b_old,
                        double *c_new, double *c_old,
                        double *mean_new, double *mean_old,
                        double *d_new, double *d_old,
                        GMRFLib_logl_tp *loglFunc_new,
                        char *loglFunc_arg_new,
                        GMRFLib_logl_tp *loglFunc_old,
                        char *loglFunc_arg_old,
                        char *fixed_value,
                        GMRFLib_graph_tp *graph,
                        GMRFLib_Qfunc_tp *Qfunc_new,
                        char *Qfunc_arg_new,
                        GMRFLib_Qfunc_tp *Qfunc_old,
                        char *Qfunc_arg_old,
                        GMRFLib_Qfunc_tp *Qfunc_old2new,
                        char *Qfunc_arg_old2new,
                        GMRFLib_Qfunc_tp *Qfunc_new2old,
                        char *Qfunc_arg_new2old,
                        GMRFLib_constr_tp *constr_new,
                        GMRFLib_constr_tp *constr_old,
                        GMRFLib_optimize_param_tp *optpar,
                        GMRFLib_blockupdate_param_tp *blockupdate_par);
```

Arguments:

`laccept (output)`: The log of the acceptance rate of the proposed (new) value of $\boldsymbol{x}$, returned by `x_new`.

`x_new (output)`: The proposed values of $\boldsymbol{x}$, denoted by superscript $\boldsymbol{x}^{'}$.

`x_old (input)`: The current (old) update of $\boldsymbol{x}$.

`b_new (input)`: The value of $\boldsymbol{b}$ in (30) based on proposed values of the hyper-parameters, $\boldsymbol{\theta}^{'}$. If `NULL`, $\boldsymbol{b}$ is assumed to be `0`.

`b_old (input)`: The value of $\boldsymbol{b}$ in (30) based on current (old) values of the hyper-parameters, $\boldsymbol{\theta}$. If `NULL`, $\boldsymbol{b}$ is assumed to be `0`.

`c_new (input)`: The value of $\boldsymbol{c}$ in (30) based on proposed values of the hyper-parameters, $\boldsymbol{\theta}^{'}$. If `NULL`, $\boldsymbol{c}$ is assumed to be `0`.

`c_old (input)`: The value of $\boldsymbol{c}$ in (30) based on current (old) values of the hyper-parameters, $\boldsymbol{\theta}$. If `NULL`, $\boldsymbol{c}$ is assumed to be `0`.

`mean_new (input)`: The value of $\boldsymbol{\mu}$ in (30) based on proposed values of the hyper-parameters, $\boldsymbol{\theta}^{'}$. If `NULL`, $\boldsymbol{\mu}$ is assumed to be `0`.

`mean_old (input)`: The value of $\boldsymbol{\mu}$ in (30) based on current (old) values of the hyper-parameters, $\boldsymbol{\theta}$. If `NULL`, $\boldsymbol{\mu}$ is assumed to be `0`.

`d_new (input)`: The value of $\boldsymbol{d}$ in (30) based on proposed values of the hyper-parameters, $\boldsymbol{\theta}^{'}$. If `NULL`, $\boldsymbol{d}$ is assumed to be `0`.

`d_old (input)`: The value of $\boldsymbol{d}$ in (30) based on current (old) values of the hyper-parameters, $\boldsymbol{\theta}$. If `NULL`, $\boldsymbol{d}$ is assumed to be `0`.

`loglFunc_new (input)`: A function of type `GMRFLib_logl_tp`, returning the value of the function $f(x_i)$ in (30), in many applications equal to the log-likelihood of the problem. If the function $f(x_i)$ depends on unknown hyper-parameters, this function uses the proposed values $\boldsymbol{\theta}^{'}$.

`loglFunc_arg_new (input)`: A `character` pointer holding the address of a variable or data structure defining additional arguments to the function `loglFunc_new`.

`loglFunc_old (input)`: A function of type `GMRFLib_logl_tp`, returning the value of the function $f(x_i)$ in (30), in many applications equal to the log-likelihood of the problem. If the function $f(x_i)$ depends on unknown hyper-parameters, this function uses the current values $\boldsymbol{\theta}$, if not, this function should be equal to `loglFunc_new`.

`loglFunc_arg_old (input)`: A `character` pointer holding the address of a variable or data structure defining additional arguments to the function `loglFunc_old`.

`fixed_value (input)`: If `!NULL`, the block-sampling of $\boldsymbol{x}$ is done conditionally on fixed values. The elements of this array, of length `graph->n`, should take the value 0 or 1. The conditional mean and covariance matrix of $\{x_i : \texttt{fixed\_value}[i] = 0\}$ given $\{x_i : \texttt{fixed\_value}[i] = 1\}$ are computed, and elements of the GMRF $\boldsymbol{x}$ corresponding to $\texttt{fixed\_value} = 1$, are kept fixed at the corresponding values specified in the argument `x_old`.

`graph (input)`: The graph on which the GMRF $\boldsymbol{x}$ is defined.

`Qfunc_new (input)`: A function of type `GMRFLib_Qfunc_tp`, computing the values of the precision matrix $\boldsymbol{Q}$. If the function depends on unknown hyper-parameters, this function computes the precision matrix as a function of their proposed values $\boldsymbol{\theta}^{'}$.

`Qfunc_arg_new (input)`: A `character` pointer holding the address of a variable or data structure defining additional arguments to the function `Qfunc_new`.

`Qfunc_old (input)`: A function of type `GMRFLib_Qfunc_tp`, computing the values of the precision matrix $\boldsymbol{Q}$. If the function depends on unknown hyper-parameters, this function computes the precision matrix as a function of their current (old) values $\boldsymbol{\theta}$. If not, this function should be equal to `Qfunc_new`.

`Qfunc_arg_old (input)`: A `character` pointer holding the address of a variable or data structure defining additional arguments to the function `Qfunc_old`.

`Qfunc_old2new (input)`: This function implements the $\boldsymbol{Q}$-matrix used when proposing a new value $\boldsymbol{x}^{'}$ given the current value $\boldsymbol{x}$. It is needed only in cases where the precision matrix depends on unknown hyper-parameters, and is different from the matrix `Qfunc_new`. If `NULL`, `Qfunc_old2new` is set equal to `Qfunc_new`.

`Qfunc_arg_old2new (input)`: A `character` pointer holding the address of a variable or data structure defining additional arguments to the function `Qfunc_old2new`.

`Qfunc_new2old (input)`: This function implements the $\boldsymbol{Q}$-matrix for the reverse of the proposal process, going from the proposed value $\boldsymbol{x}^{'}$ to the the current value $\boldsymbol{x}$. It is needed only in cases where the precision matrix depends on unknown hyper-parameters, and is different from the matrix `Qfunc_old`. If `NULL`, `Qfunc_new2old` is set equal to `Qfunc_old`.

`Qfunc_arg_new2old (input)`: A `character` pointer holding the address of a variable or data structure defining additional arguments to the function `Qfunc_new2old`.

`constr_new (input)`: A pointer to a `GMRFLib_constr_tp`-object holding the information on the type of linear constraint, in the case of constrained sampling. If the parameters of the constraint depend on unknown hyper-parameters, `constr_new` should define the constraint corresponding to the proposed values $\boldsymbol{\theta}^{'}$.

`constr_old (input)`: A pointer to a `GMRFLib_constr_tp`-object holding the information on the type of linear constraint, in the case of constrained sampling. If the parameters of the constraint depend on unknown hyper-parameters, `constr_old` should define the constraint corresponding to the current values $\boldsymbol{\theta}$. If the constraint does not depend on the hyper-parameters, `constr_old` and `constr_new` should be equal.

`optpar (input)`: The options to the optimizer. If `NULL`, default options are used.

`blockupdate_par (input)`: Some specifications for the block-sampler, as a `GMRFLib_blockupdate_param_tp`-object.

Details: All parameters in (30) might depend on unknown hyper-parameters $\boldsymbol{\theta}$, such that $\boldsymbol{b} = \boldsymbol{b}(\theta)$ and so on. These hyper-parameters are also to be updated by the MCMC algorithm, and the contributions to the acceptance rate from these parameters are to be added to `laccept` after invoking `GMRFLib_blockupdate`. If the parameters in (30) depend on $\boldsymbol{\theta}$, two sets of parameters have to be provided, one depending on the current (old) value of $\boldsymbol{\theta}$, and one depending on the proposed values $\boldsymbol{\theta}^{'}$, that are assumed to be generated prior to calling `GMRFLib_blockupdate`. The two sets of values take subscripts `new` (corresponding to the proposals $\boldsymbol{\theta}^{'}$) and `old` (corresponding to the current, or old, values $\boldsymbol{\theta}$). The same comment applies to the set of function defining the precision matrix $\boldsymbol{Q}$, these functions might also depend on hyper-parameters.

For the Q-functions, two additional functions are provided. These are needed for generating the proposals from the Gaussian proposal distribution in cases where the precision matrix depend on unknown hyper-parameters, and the precision matrix of the proposal distribution is different from the precision matrix $\boldsymbol{Q}$ of the GMRF at input. This might be the case when $\boldsymbol{Q}(\boldsymbol{\theta})+\text{diag}(\boldsymbol{c})$ is singular. The function `GMRFLib_Qfunc_old2new` is the $\boldsymbol{Q}$-matrix for the distribution generating a proposal $\boldsymbol{x}^{'}$ given $\boldsymbol{x}$, and `GMRFLib_Qfunc_new2old` is the corresponding function, implementing the $\boldsymbol{Q}$-matrix used in the reverse process, going from $\boldsymbol{x}^{'}$ to $\boldsymbol{x}$.

Example: See Section 5.4.

See also: `GMRFLib_logl_tp`

GMRFLib_logl_tp

Description: A template for routines computing the functions $f(x_i)$ of (30). In many applications, these functions are equal to the elements of the log-likelihood of the GMRF $\boldsymbol{x}$, given the data.

Declaration:

```
typedef int GMRFLib_logl_tp(double *logll, double *x_i, int m, int idx,
                            double *x_vec, char *logl_arg);
```

Arguments:

`logll (output)`: At output, `logll` is a length `m` array holding the values of $f(x_i)$, computed for `m` different values of element `idx` of $\boldsymbol{x}$.

`x_i (input)`: A length `m` array of values of $x_{\texttt{idx}}$, for which to compute $f(x_{\texttt{idx}})$.

`m (input)`: The number of values of $x_{\texttt{idx}}$ for which to compute the function values.

`idx (input)`: The index of $\boldsymbol{x}$ corresponding to the values in `x_i`.

`x_vec (input)`: A length `n` array holding the current values of all elements of the GMRF $\boldsymbol{x}$. The value of `x_vec[idx]` is substituted by each element of `x_i` in turn.

`logl_arg (input)`: Additional arguments to the function. The `logl_arg` `character`-pointer is to hold the address of a variable or data structure holding the additional arguments to the function, including the data.

Return value: The function is to return an error code (or 0).

Example: The function `loglik` computes the Poisson log-likelihood $y_{\texttt{idx}} \sim \text{Pois}(E_{\texttt{idx}} \exp(x_{\texttt{idx}}))$.

```
int loglik(double *logll, double *x, int m, int idx,
           double *x_vec, char *arg)
{
    int i;
    double *y, *E;
    char **args;

    /*
       the data y and E come through the arg-pointer
     */
    args  = (char **)arg;
    y     = (double *)args[0];
    E     = (double *)args[1];

    /*
       compute the log-likelihood
     */
    for(i=0;i<m;i++) logll[i] = y[idx]*x[i]-E[idx]*exp(x[i]);

    return 0;
}
```

See also: `GMRFLib_blockupdate`

4.5 Error-handling

If an error occurs running the routines from `GMRFLib`, an error message will be issued stating the type of error, and the function, the file and the line at which the error has occurred. By default, the program is core dumped and aborted at the occurrence of an error, using the function `GMRFLib_error_handler` defined in Section 4.5.2. This side effect can be overridden by the user, re-defining the error handling function as described in Section 4.5.3.

4.5.1 Error messages

When an error occurs in one of the `GMRFLib`-routines, one out of 17 error messages is issued. The error numbers and the corresponding error messages are listed in Table 4.5.1.

Error number	Error message
0	No error, please ignore
1	Malloc failed
2	Matrix is not positive definite
3	Matrix is singular
4	Invalid argument
5	Graph is invalid
6	Error while reordering the graph
7	Error while reading graph from file
8	Error opening file
9	Invalid value of parameter
10	No index found
11	Pointer is required to be non-NULL
12	The Newton-Raphson optimizer did not converge
13	The Conjugate-Gradient optimizer did not converge
14	The line-optimizer in the Conjugate-Gradient optimizer did not converge
15	Error occurred in the dcdf-library
16	(((this is an unknown error code)))

Table 8: The error numbers and error messages issued in the `GMRFLib`-library routines.

4.5.2 Error handler

GMRFLib_error_handler_tp

Description: A template function declaration for specifying error-handling functions.

Declaration:

```
typedef int GMRFLib_error_handler_tp (const char *reason,
                                       const char *file,
                                       const char *function, int line,
                                       const char *id, int errorno,
                                       const char *condition)
```

Arguments:

- `reason`: A character string describing the reason for the error (See Table 4.5.1 for the list of error messages included in `GMRFLib`).
- `function`: The name of the function in which the error occurs.
- `file`: The file name.
- `line`: The line number of the file.
- `id`: The RCS Id (version control Id) of the file.
- `errorno`: The error id number (see Table 4.5.1).
- `condition`: If a condition in a routine, like `if (i>0)`, is violated, the condition can be transfered using this argument.

Return value: The function is to return the error number `errorno`.

See also: `GMRFLib_error_handler`,`GMRFLib_set_error_handler`

GMRFLib_error_handler

Description: The default error handling function, of type `GMRFLib_error_handler_tp`.

Declaration:

```
int GMRFLib_error_handler (const char *reason, const char *file,
                           const char *function, int line,
                           const char *id, int errorno,
                           const char *condition)
```

Arguments: See `GMRFLib_error_handler_tp`.

Return value: The error number. The function is aborted if reporting an error, so a value is returned only if `errorno` $= 0$.

Details: This is the default error handling function integrated in the library routines. It is called by the function `GMRFLib_handle_error`. When an error occurs, the function prints an error message of the form

```
        GMRFLib has recived error no [9]
        Reason     : Invalid value of parameter
        Caused by  : if (!(argc>1)) error_handler()
        Function   : main
        File       : test-graph.c
        Line       : 25
        RCSId      : $Id: GMRFLib-manual.tex,v 1.6 2002/01/06 19:49:48 hrue Exp hrue $

Abort trap (core dumped)
```

and the program is aborted, with a core dump. In the example above, the error message states that the command line arguments of the main program on the file `test-graph.c` is missing. To avoid aborting the program, a new error handling function can be defined, returning the error code without aborting. To change to another error handling function, use `GMRFLib_set_error_handler`.

See also: `GMRFLib_error_handler_tp`, `GMRFLib_handle_error`, `GMRFLib_set_error_handler`.

GMRFLib_error_reason

Description: Returns the type of error, given the error code `errorno`.

Declaration:

```
const char *GMRFLib_error_reason(int errorno)
```

Arguments:

`errorno`: The error number (See Table 4.5.1)

Return value: A character string, giving the error message corresponding to error number `errorno` (See Table 4.5.1).

Details: This function is used within the library generating the `reason` argument in the default error handling function `GMRFLib_error_handler`.

See also: `GMRFLib_error_handler`.

GMRFLib_handle_error

Description: Handles the error by calling an internal macro, issuing an error message. This is the function that is actually called at the occurrence of an error.

Declaration:

```
int GMRFLib_handle_error(const char *file, const char *function,
                         int line, const char *id, int errorno,
                         const char *condition)
```

Arguments: See `GMRFLib_error_handler_tp`.

See also: `GMRFLib_error_handler_tp`, `GMRFLib_error_handler`, `GMRFLib_set_error_handler`

4.5.3 Overriding the default error messages

The default error messages can be overridden by the user by defining a new error handling function of a specified format, `GMRFLib_error_handler_tp`, described above. Within the library, a static function pointer `handler` is defined, pointing to a function of type `GMRFLib_error_handler_tp`. By default, this is `NULL`, indicating that the default error handling function, `GMRFLib_error_handler` is to be used. To override the default function, `handler` should be set to point at the user-specified `GMRFLib_error_handler_tp`-function, which can be done using the library function `GMRFLib_set_error_handler`.

GMRFLib_set_error_handler

Description: Sets the error handler function to be the default function or a user specified function.

Declaration:

```
int GMRFLib_set_error_handler(GMRFLib_error_handler_tp *new_error_handler)
```

Arguments:

`new_error_handler`: A pointer to a user-specified `GMRFLib_error_handler_tp`-function. If `new_error_handler` = `NULL`, the default error handler function is chosen.

Details: To override the error handling function, the user should write a function with return value and argument list as defined by the `GMRFLib_error_handler_tp`-function template, and then call `GMRFLib_set_error_handler`. As a result, the user-specified function will be called at the occurrence of an error. For convenience, the implementation of the default function `GMRFLib_error_handler` is given in Appendix A.

See also: `GMRFLib_error_handler_tp`, `GMRFLib_error_handler`

4.6 Random number generation

By default, the sampling routines in `GMRFLib` use a built-in random generator provided with the library. The random generator includes four functions, which can be overloaded by the user. The functions are accessed by a set of `external` function pointers, declared by

```
extern GMRFLib_uniform_tp          *GMRFLib_uniform;
extern GMRFLib_uniform_init_tp     *GMRFLib_uniform_init;
extern GMRFLib_uniform_getstate_tp *GMRFLib_uniform_getstate;
extern GMRFLib_uniform_setstate_tp *GMRFLib_uniform_setstate;
```

By default, the function pointers are assigned to the random generator functions provided by `GMRFLib`. The function type templates are documented in the next subsection. To override the default generator, the user should re-implement ALL four functions, and reset the global function pointers to point to these user-defined functions. This is done in the C-code by including the following definitions:

```
GMRFLib_uniform = my_uniform;
GMRFLib_uniform_init = my_uniform_init;
GMRFLib_uniform_getstate = my_uniform_getstate;
GMRFLib_uniform_setstate = my_uniform_setstate;
```

Here, the functions `my_*` are the user-defined functions of type `GMRFLib_uniform_tp`, `GMRFLib_uniform_init_tp`, `GMRFLib_uniform_getstate_tp` and `GMRFLib_uniform_setstate_tp` respectively, all declared below.

4.6.1 Template function declarations

`GMRFLib_uniform_tp`

Description: A template for functions generating a random number form the Uniform$(0, 1)$ - distribution.

Declaration:

```
typedef double GMRFLib_uniform_tp(void);
```

Arguments: None.

Return value: The function is to return a sample from the Uniform$(0, 1)$ - distribution.

See also: `GMRFLib_uniform_init_tp`, `GMRFLib_uniform_getstate_tp`, `GMRFLib_uniform_setstate_tp`.

`GMRFLib_uniform_init_tp`

Description: A template for initializing the random number generator, for generating Uniform$(0, 1)$ random numbers.

Declaration:

```
typedef int GMRFLib_uniform_init_tp(unsigned int seed);
```

Arguments:

seed: The seed of the random number generator.

Return value: An integer, possibly an error code.

See also: GMRFLib_uniform_tp, GMRFLib_uniform_getstate_tp, GMRFLib_uniform_setstate_tp.

GMRFLib_uniform_getstate_tp

Description: A template for extracting the state of the random number generator.

Declaration:

```
typedef char*  GMRFLib_uniform_getstate_tp(void);
```

Arguments: None.

Return value: A character array, defining the state of the random number generator.

Details: This functions is provided such that the state of the random generator can be extracted within the sampling routines. To reset the generator using the extracted state, a function of type GMRFLib_uniform_setstate_tp is used.

See also: GMRFLib_uniform_tp, GMRFLib_uniform_init_tp, GMRFLib_uniform_setstate_tp.

GMRFLib_uniform_setstate_tp

Description: A template for re-setting the state of the random number generator.

Declaration:

```
typedef int    GMRFLib_uniform_setstate_tp(char *state);
```

Arguments: A character array, defining the state of the random number generator.

Return value: An integer, possibly an error code.

Details: This functions is used in connection with a function of type GMRFLib_uniform_getstate_tp, to reset the state of the random number generator within a sampling program.

See also: GMRFLib_uniform_tp, GMRFLib_uniform_init_tp, GMRFLib_uniform_getstate_tp.

5 Examples

5.1 Graph specification and handling

5.1.1 Reading and printing graphs

Description:

This program performs the following tasks:

- Reads a graph from the file `graph1.dat`, given below, of the form required by the function `GMRFLib_read_graph`. The specified graph has 10 nodes, with from 1 to 5 neighbors.

```
10
0 3 1 2 3
1 3 0 2 3
2 5 0 1 3 4 5
3 4 0 1 2 5
4 3 2 5 7
5 5 2 3 4 6 7
6 2 5 7
7 4 4 5 6 8
8 2 7 9
9 1 8
```

- Prints the graph specification to standard output.
- Computes a subgraph, by specifying nodes to be removed from the graph. More specifically, the first half of the nodes, that is node $0, \ldots, 4$, are specified to be removed. To compute the subgraph, the function `GMRFlib_compute_subgraph` is called.
- Frees allocated memory.

Program code:

```
#include "GMRFLib/GMRFLib.h"

#define Calloc(n,type) (type *)calloc((size_t)(n),sizeof(type))

int main(int argc,  char **argv)
{
    GMRFLib_graph_tp *graph;       /* Graph object */
    GMRFLib_graph_tp *sgraph;      /* Subgraph     */
    char *gfile = "graph1.dat";    /* Input file   */
    char *remove;
    int i;

    /* Read graph from the file  gfile,                      */
    /* and print the graph specification to standard output */
    printf("\nReading a graph from the file '%s':\n\n", gfile);

    GMRFLib_read_graph(&graph, gfile);
    GMRFLib_print_graph(stdout, graph);

    /* Indicate nodes that are to be removed */
```

```
    remove = Calloc(graph->n, char);
    for (i=0;i<graph->n/2;i++)
        remove[i] = 1;

    /* Compute and print subgraph */
    printf("\nComputing subgraph:\n\n");
    GMRFLib_compute_subgraph(&sgraph, graph, remove);
    GMRFLib_print_graph(stdout, sgraph);

    GMRFLib_free_graph(graph);
    GMRFLib_free_graph(sgraph);

    return 0;
}
```

Output:

```
Reading a graph from the file 'graph1.dat':

graph has 10 nodes and bandwidth 3
node 0 has 4 neighbors: 0 1 2 5
node 1 has 3 neighbors: 0 2 3
node 2 has 5 neighbors: 0 1 3 4 5
node 3 has 3 neighbors: 0 1 2
node 4 has 3 neighbors: 2 5 7
node 5 has 5 neighbors: 0 2 4 6 7
node 6 has 2 neighbors: 5 7
node 7 has 4 neighbors: 4 5 6 8
node 8 has 2 neighbors: 7 9
node 9 has 1 neighbors: 8
node 0 appears after reordering in position 2
node 1 appears after reordering in position 1
node 2 appears after reordering in position 3
node 3 appears after reordering in position 0
node 4 appears after reordering in position 5
node 5 appears after reordering in position 4
node 6 appears after reordering in position 6
node 7 appears after reordering in position 7
node 8 appears after reordering in position 8
node 9 appears after reordering in position 9

Computing subgraph:

graph has 5 nodes and bandwidth 2
node 0 has 2 neighbors: 1 2
node 1 has 2 neighbors: 0 2
node 2 has 3 neighbors: 0 1 3
node 3 has 2 neighbors: 2 4
node 4 has 1 neighbors: 3
node 0 appears after reordering in position 4
node 1 appears after reordering in position 3
node 2 appears after reordering in position 2
node 3 appears after reordering in position 1
node 4 appears after reordering in position 0
node 0 corresponds to mother-node 5
node 1 corresponds to mother-node 6
```

```
node 2 corresponds to mother-node 7
node 3 corresponds to mother-node 8
node 4 corresponds to mother-node 9
```

5.1.2 Creating lattice graphs, linear graphs and folded graphs

Description:

This program describes how to create a lattice graph and a linear graph, and how to expand the neighbourhood of a graph. The lattice graph is defined on a 6×7 lattice, such that $n_{\text{row}} = 6$ and $n_{\text{col}} = 7$. The neighbourhood is defined to be a 3×3-neighbourhood, such that $m_{\text{row}} = m_{\text{col}} = 1$ in (1). The node numbers corresponding to the lattice indices are extracted, and the reverse operation is also illustrated. The linear graph is defined by a cyclic $AR(1)$-model with 10 elements. A third graph is generated by expanding the neighbourhood of the linear graph twice.

Program code:

```
#include "GMRFLib/GMRFLib.h"

int main(int argc,  char **argv)
{
    GMRFLib_graph_tp *lgraph;       /* Graph object, lattice graph */
    GMRFLib_graph_tp *argraph;      /* Graph object, linear graph */
    GMRFLib_graph_tp *ngraph;       /* Graph object, folded graph */
    int i, j, nindx, li, lj;
    const int nrow=6, ncol=7;       /* Size of lattice */

    /* Create a non-cyclic graph on a 6 by 7 lattice, */
    /* using a 3 by 3 neighborhood */
    printf("\nCreating a lattice graph:\n\n");

    GMRFLib_create_lattice(&lgraph, nrow, ncol, 1, 1, 0);
    GMRFLib_print_graph(stdout, lgraph);

    /* Extract the ordering of the nodes of the lattice, */
    /* and recompute the lattice indices from the node indices */

    printf("\nExtracting node indices:\n\n");
    for(i=0;i<nrow;i++)
        for(j=0;j<ncol;j++)
        {
            GMRFLib_lattice2node(&nindx, i, j, nrow, ncol);
            printf("Lattice indices (%d,%d) correspond to node no. %d\n", i, j, nindx);
        }
    printf("\nExtracting lattice indices:\n\n");
    for (i=0;i<lgraph->n;i++)
    {
        GMRFLib_node2lattice(i, &li, &lj, nrow, ncol);
        printf("Node no. %d corresponds to lattice indices (%d,%d)\n", i, li, lj);
    }


    /* Create a cyclic linear graph on an AR(1)-model */
    printf("\nCreating a linear graph:\n\n");
```

```
    GMRFLib_make_linear_graph(&argraph, 10, 1, 1);
    GMRFLib_print_graph(stdout, argraph);


    /* Expand the neighborhood of linear graph twice */
    printf("\nCreating a folded graph:\n\n");
    GMRFLib_nfold_graph(&ngraph, argraph, 2);
    GMRFLib_print_graph(stdout, ngraph);

    GMRFLib_free_graph(lgraph);
    GMRFLib_free_graph(argraph);
    GMRFLib_free_graph(ngraph);

    return 0;
}
```

Output:

```
Creating a lattice graph:

graph has 42 nodes and bandwidth 8
node 0 has 3 neighbors: 1 7 8
node 1 has 5 neighbors: 0 2 7 8 9
node 2 has 5 neighbors: 1 3 8 9 10
node 3 has 5 neighbors: 2 4 9 10 11
node 4 has 5 neighbors: 3 5 10 11 12
node 5 has 5 neighbors: 4 6 11 12 13
node 6 has 3 neighbors: 5 12 13
node 7 has 5 neighbors: 0 1 8 14 15
node 8 has 8 neighbors: 0 1 2 7 9 14 15 16
node 9 has 8 neighbors: 1 2 3 8 10 15 16 17
  .
  .
  .
node 0 appears after reordering in position 0
node 1 appears after reordering in position 1
node 2 appears after reordering in position 2
node 3 appears after reordering in position 3
  .
  .
  .

Extracting node indices:

Lattice indices (0,0) correspond to node no. 0
Lattice indices (0,1) correspond to node no. 1
Lattice indices (0,2) correspond to node no. 2
Lattice indices (0,3) correspond to node no. 3
Lattice indices (0,4) correspond to node no. 4
Lattice indices (0,5) correspond to node no. 5
Lattice indices (0,6) correspond to node no. 6
Lattice indices (1,0) correspond to node no. 7
Lattice indices (1,1) correspond to node no. 8
  .
  .
  .
```

```
Extracting lattice indices:

Node no. 0 corresponds to lattice indices (0,0)
Node no. 1 corresponds to lattice indices (0,1)
Node no. 2 corresponds to lattice indices (0,2)
Node no. 3 corresponds to lattice indices (0,3)
Node no. 4 corresponds to lattice indices (0,4)
Node no. 5 corresponds to lattice indices (0,5)
Node no. 6 corresponds to lattice indices (0,6)
Node no. 7 corresponds to lattice indices (1,0)
Node no. 8 corresponds to lattice indices (1,1)
  .
  .
  .


Creating a linear graph:

graph has 10 nodes and bandwidth 2
node 0 has 2 neighbors: 1 9
node 1 has 2 neighbors: 0 2
node 2 has 2 neighbors: 1 3
node 3 has 2 neighbors: 2 4
node 4 has 2 neighbors: 3 5
node 5 has 2 neighbors: 4 6
node 6 has 2 neighbors: 5 7
node 7 has 2 neighbors: 6 8
node 8 has 2 neighbors: 7 9
node 9 has 2 neighbors: 0 8
node 0 appears after reordering in position 0
node 1 appears after reordering in position 1
node 2 appears after reordering in position 3
node 3 appears after reordering in position 5
node 4 appears after reordering in position 7
node 5 appears after reordering in position 9
node 6 appears after reordering in position 8
node 7 appears after reordering in position 6
node 8 appears after reordering in position 4
node 9 appears after reordering in position 2

Creating a folded graph:

graph has 10 nodes and bandwidth 5
node 0 has 4 neighbors: 1 2 8 9
node 1 has 4 neighbors: 0 2 3 9
node 2 has 4 neighbors: 0 1 3 4
node 3 has 4 neighbors: 1 2 4 5
node 4 has 4 neighbors: 2 3 5 6
node 5 has 4 neighbors: 3 4 6 7
node 6 has 4 neighbors: 4 5 7 8
node 7 has 4 neighbors: 5 6 8 9
node 8 has 4 neighbors: 0 6 7 9
node 9 has 4 neighbors: 0 1 7 8
node 0 appears after reordering in position 0
node 1 appears after reordering in position 1
node 2 appears after reordering in position 2
node 3 appears after reordering in position 5
```

```
node 4 appears after reordering in position 6
node 5 appears after reordering in position 9
node 6 appears after reordering in position 7
node 7 appears after reordering in position 8
node 8 appears after reordering in position 3
node 9 appears after reordering in position 4
```

5.2 Sampling from a GMRF

In the example below, unconditional and conditional sampling from a GMRF $\boldsymbol{x}$ are illustrated. Th unconditional density of the GMRF is given by the general expression in (2), re-stated below for convenience:

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T(\boldsymbol{Q}+\text{diag}(\boldsymbol{c}))(\boldsymbol{x}-\boldsymbol{\mu})+\boldsymbol{b}^T\boldsymbol{x}\right). \tag{37}$$

For all sampling methods illustrated in the example, we define the graph to be a lattice graph on a 6×6 lattice, using a 3×3 neighbourhood. The $\boldsymbol{Q}$-matrix is defined by

$$\boldsymbol{Q} = \kappa \begin{pmatrix} \texttt{nnbs}[1] & -1 & 0 & 0 & \cdots & 0 \\ -1 & \texttt{nnbs}[2] & -1 & 0 & \cdots & 0 \\ 0 & -1 & \texttt{nnbs}[3] & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \texttt{nnbs}[6] \end{pmatrix}, \tag{38}$$

where `nnbs`[i] is the number of neighbours of node i. This matrix is singular, but the singularity is resolved either by adding elements to the diagonal, or conditioning on fixed values. In the example, we use $\kappa = 1$.

Four sampling methods are implemented in the example. These are

1. *Unconditional sampling*. In this case, we generate samples from (37). The remaining parameters are set to $\boldsymbol{b} = \boldsymbol{0}$, $\boldsymbol{\mu} = \boldsymbol{0}$ and $\boldsymbol{c} = \boldsymbol{1}_n$, The initial values of the GMRF (that are not used in this case) are set to $\boldsymbol{x} = \boldsymbol{0}$.

2. *Conditioning on fixed values*. We sample from $\boldsymbol{x}_{-\mathcal{F}}|\boldsymbol{x}_{\mathcal{F}}$, where $\mathcal{F}$ is the set of indices i for which x_i is fixed. In the example we let $\mathcal{F} = \{i : i \mod 2 = 0\}$, and $x[i] = \text{Unif}(0,1);\ i = 0, \ldots, n-1$. The remaining parameters are assigned the values $\boldsymbol{b} = \boldsymbol{0}$, $\boldsymbol{\mu} = \boldsymbol{0}$ and $\boldsymbol{c} = \boldsymbol{0}$.

3. *Conditioning on a deterministic linear constraint*, sampling from $\pi(\boldsymbol{x}|\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e})$. In the example, we use the constraints $\sum_i x_i = 0$ and $x_0 + 2x_1 = 1$, which correspond to

$$\boldsymbol{A} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 0 & \cdots & 0 \end{pmatrix}, \text{ and } \boldsymbol{e} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{39}$$

   Also, we condition on fixed values, fixing the same elements of $\boldsymbol{x}$ as for method 2. The values of the GMRF are $\boldsymbol{x} = \text{Unif}(0,1);\ i = 0, \ldots, n-1$. The remaining parameters are set to $\boldsymbol{b} = \text{Unif}(0,1)$, $\boldsymbol{\mu} = \boldsymbol{0}$ and $\boldsymbol{c} = \boldsymbol{1}_n$.

4. *Conditioning on a stochastic linear constraint*, sampling from $\pi(\boldsymbol{x}|\boldsymbol{A}\boldsymbol{x} = \boldsymbol{e} + \boldsymbol{\epsilon})$. This is equivalent to sampling unconditionally from (11). We use the constraint $\sum_i x_i = 0$ and assume that $\epsilon \sim N(0, \sigma^2)$, where $\sigma^2 = 1$. This corresponds to

$$\boldsymbol{A} = \begin{pmatrix} 1 & 1 & \cdots & 1 \end{pmatrix}, \ \boldsymbol{e} = e = 0 \text{ and } \boldsymbol{\Sigma}_\epsilon = \sigma^2. \tag{40}$$

   The other parameters are assign values equal to the values used for method 3, but we do not condition on fixed values in this case.

Program code:

```
#include "GMRFLib/GMRFLib.h"

#define Calloc(n,type) (type *)calloc((size_t)(n),sizeof(type))
#define FREE(ptr) {if(ptr)free((void *)(ptr));ptr=NULL;}

/* The Q-function, returning
   return Q(i,j), i=node, j=nnode, corresponding to
   pi(x)  \exp(-\frac{1}{2} \sum_{i\sim j} (x_i-x_j)^2)
*/

double Qfunc(int node, int nnode, char *arg)
{
    GMRFLib_graph_tp *g = (GMRFLib_graph_tp *) arg;
    if (node != nnode)
        return -1.0;
    else
        return g->nnbs[node];
}


int main(int argc,  char **argv)
{
    GMRFLib_graph_tp *graph;
    GMRFLib_problem_tp *problem;
    GMRFLib_constr_tp *constr = NULL;

    double *x, *b, *c;
    int i, method, seed;
    char *argQfunc, *fixed=NULL;
    FILE* fp;

    const int nrow=6, ncol=6;  /* Size of lattice graph */

    if (argc<3)
    {
        printf("Usage: %s SEED METHOD\n", argv[0]);
        exit(-1);
    }
    seed = atoi(argv[1]);
    method = atoi(argv[2]);

    /* Initialize random number generator: */
    GMRFLib_uniform_init(seed);

    /* create a lattice graph */
    GMRFLib_create_lattice(&graph, nrow, ncol, 1, 1, 0);


    x = Calloc(graph->n, double);
    c = Calloc(graph->n, double);
    b = Calloc(graph->n, double);


    /* Unconditional sampling: */
    if (method==1)
    {
```

```
    for (i=0;i<graph->n;i++)
    {
        x[i] = 0.0;
        c[i] = 1.0;
    }

    argQfunc = (char*) graph;

    GMRFLib_init_problem(&problem, x, NULL, c, NULL,
                         graph, Qfunc, argQfunc, NULL, NULL, 0);
}

/* Conditional sampling, conditional on fixed values: */
if (method==2)
{
    fixed = Calloc(graph->n, char);
    for (i=0;i<graph->n;i=i+2)
    {
        fixed[i] = 1;
    }

    for (i=0;i<graph->n;i++)
    {
        x[i] = GMRFLib_uniform();
        c[i] = 0.0;
    }
    argQfunc = (char*) graph;
    GMRFLib_init_problem(&problem, x, NULL, c, NULL,
                         graph, Qfunc, argQfunc, fixed, NULL, 0);
}

/* Conditional sampling, conditional on a linear deterministic constraint: */
if (method==3)
{
    fixed = Calloc(graph->n, char);
    for (i=0;i<graph->n;i=i+2)
    {
        fixed[i] = 1;
    }
    for (i=0;i<graph->n;i++)
    {
        x[i] = GMRFLib_uniform();
        c[i] = 1.0;
        b[i] = GMRFLib_uniform();
    }

    /* Create constraint: */
    GMRFLib_create_constr(&constr);
    constr->nc = 2;
    constr->a_matrix = Calloc(constr->nc*graph->n, double);
    constr->e_vector = Calloc(constr->nc, double);

    constr->e_vector[0] = 0.0;
    constr->e_vector[1] = 0.0;

    constr->a_matrix[1] = 1.0;
    constr->a_matrix[3] = 0.0;
    constr->a_matrix[5] = 2.0;
```

```
    for (i=0;i<graph->n;i++)
    {
        constr->a_matrix[i*constr->nc] = 1.0;
        if (i>2) constr->a_matrix[i*constr->nc+1] = 0.0;
    }
    GMRFLib_prepare_constr(constr);

    argQfunc = (char*) graph;
    GMRFLib_init_problem(&problem, x, b, c, NULL,
                         graph, Qfunc, argQfunc, fixed, constr, 0);
}

/* Conditional sampling, conditional on a linear stochastic constraint: */

if (method==4)
{
    for (i=0;i<graph->n;i++)
    {
        x[i] = GMRFLib_uniform();
        c[i] = 1.0;
        b[i] = GMRFLib_uniform();
    }

    /* Create constraint: */
    GMRFLib_create_constr(&constr);
    constr->nc = 1;
    constr->a_matrix = Calloc(graph->n, double);
    for(i=0;i<graph->n;i++)
        constr->a_matrix[i] = 1.0;
    constr->e_vector = Calloc(1, double);
    constr->e_vector[0] = 0.0;
    constr->errcov_diagonal = Calloc(constr->nc, double);
    constr->errcov_diagonal[0] = 1.0;
    GMRFLib_prepare_constr(constr);

    argQfunc = (char*) graph;
    GMRFLib_init_problem(&problem, x, b, c, NULL,
                         graph, Qfunc, argQfunc, NULL, constr, 0);
}

/* Sample: */
GMRFLib_sample(problem);


/* Extracting and printing the sample, */
/* the log-density and the mean: */

fp = fopen("samples.dat", "w");
for (i=0;i<graph->n;i++)
    fprintf(fp, "Sample %d: %12.6f\n", i, problem->sample[i]);
fclose(fp);

printf("Log-density of sample: %.12f\n\n", problem->sub_logdens);
printf("Sub-index mean (unconstr) mean (constr):\n");
for (i=0;i<problem->sub_graph->n;i++)
{
    printf("%d\t%12.6f\t%12.6f\n", problem->map[i],
           problem->sub_mean[i], problem->sub_mean_constr[i]);
```

```
    }

    /* Free allocated memory: */
    FREE(fixed);
    FREE(x);
    FREE(c);
    FREE(b);

    GMRFLib_free_constr(constr);
    GMRFLib_free_problem(problem);
    GMRFLib_free_graph(graph);

    return 0;
}
```

Output for `method=3`**:**

```
Log-density of sample: -9.537986860412

Sub-index mean (unconstr) mean (constr):
1            0.244652       -0.470394
3            0.420537       -0.294509
5            0.348750       -0.829304
7            0.325922       -0.228238
9            0.399286       -0.154874
11           0.394796       -0.581305
13           0.398828       -0.137457
15           0.420460       -0.115824
17           0.279265       -0.663178
19           0.484196       -0.052088
21           0.475638       -0.060646
23           0.384558       -0.557885
25           0.380659       -0.173501
27           0.370326       -0.183834
29           0.437764       -0.538338
31           0.384011       -0.331034
33           0.288129       -0.426917
35           0.534661       -0.643393
```

5.3 A worked out example smoothing a time-series data, using the *wa_problem* feature

In this example, we will illustrate the *wa_problem* feature, which is quite useful in situations like the following. Consider the time-series data $\boldsymbol{y}$ in Figure 1, which we assume to be observed according to the model

$$y_i = x_i + \epsilon_i, \tag{41}$$

where $\boldsymbol{x}$ is a hidden smooth curve and $\boldsymbol{\epsilon}$ is iid Gaussian noise with zero mean and variance 0.25^2.

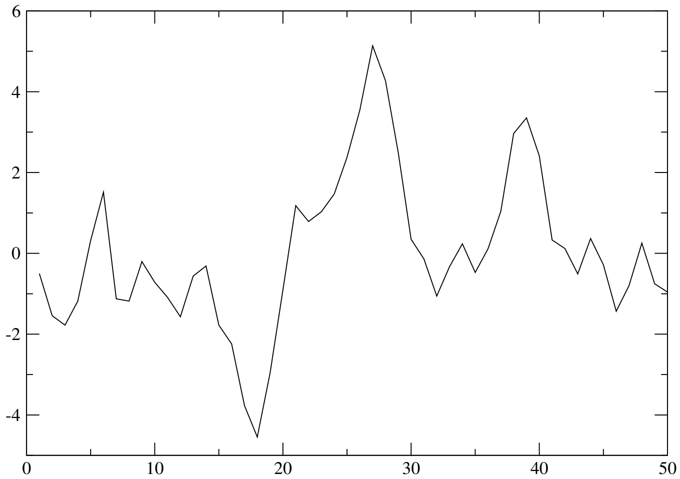

Figure 1: The observed time-series data.

As a prior, or a smoothing-prior, for $\boldsymbol{x}$, we will consider two models: a first order model

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{h}{2}\sum_{i=2}^{n}(x_i - x_{i-1})^2\right) \tag{42}$$

and a second order model

$$\pi(\boldsymbol{x}) \propto \exp\left(-\frac{h}{2}\sum_{i=3}^{n}(x_i - 2 * x_{i-1} + x_{i-2})^2\right). \tag{43}$$

The quantities of interest are the posterior mean for $\boldsymbol{x}|\boldsymbol{y}$ and samples from the posterior

$$\pi(\boldsymbol{x}|\boldsymbol{y}) \propto \exp\left(-\frac{h}{2}\sum_{i=2}^{n}(x_i - x_{i-1})^2 + \sum_{i=1}^{n}(y_i x_i/\sigma^2 + x_i^2/\sigma^2)\right), \tag{44}$$

and similar values for the second order model.

To implement Equation (44), we observe that the prior is of the *wa_problem* form (8), for which we use the function `GMRFLib_init_wa_problem` to convert the prior into the standard $\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x}$-form. The data will define the $\boldsymbol{b}$ and $\boldsymbol{c}$ vector in `GMRFLib_init_problem`.

Program code:

```
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#if !defined(__FreeBSD__)
#include <malloc.h>
#endif

/*
   include the required header-file
 */
#include "GMRFLib/GMRFLib.h"

/*
   some useful macros
 */

#define Calloc(n,type) (type *)calloc((size_t)(n),sizeof(type))
#define FREE(ptr) {if(ptr)free((void *)(ptr));ptr=NULL;}
#define MAX(a,b)  ((a) > (b) ? (a) : (b))
#define MIN(a,b)  ((a) < (b) ? (a) : (b))
#define SQR(x)    ((x)*(x))

#define DATAFILE "wa-example.dat"
#define N 50
#define NSAMPLES 10

double wafunc(int i, int j, char *arg)
{
    /*
       return the weights w_ij, including the precision sqrt(h)
     */
    double val, h;
    char **args;
    int order;

    args  = (char **)arg;
    order = *(int *)args[0];
    h     = *(double *)args[1];

    if (j > i || i < order) return 0.0;

    switch(order)
    {
    case 1:
        return ((j == i || j == i-1) ? sqrt(h) : 0.0);
    case 2:
        switch(i-j)
        {
        case 0:  val =  1.0; break;
        case 1:  val =  2.0; break;
        case 2:  val = -1.0; break;
        default: val =  0.0;
        }
        return val*sqrt(h);
```

```
    default:
        abort();
    }
    return 0.0;
}
int main(int argc, char **argv)
{
    int seed, i, k, order;
    double h = 100.0, *x, *c, *b, *y, *samples, noise_variance;
    FILE *fp;
    char *args[2];
    GMRFLib_graph_tp *wagraph;
    GMRFLib_wa_problem_tp *waproblem;
    GMRFLib_problem_tp *problem;

    if (argc < 4)
    {
        printf("Usage: %s SEED ORDER H\n", *argv);
        return 0;
    }

    /*
       get the seed and init the random generator in GMRFLib,
       read other arguments
     */
    seed = atoi(argv[1]);
    (*GMRFLib_uniform_init)(seed);

    order = atoi(argv[2]);
    h     = atof(argv[3]);

    /*
       setup arguments to wafunc
     */
    args[0] = (char *)&order;
    args[1] = (char *)&h;

    /*
       create wa-problem, note that the graph, Qfunc and Qfunc_arg
       is then provided in 'waproblem'!
     */
    GMRFLib_make_linear_graph(&wagraph, N, order, 0);
    GMRFLib_init_wa_problem(&waproblem, wagraph, wafunc, (char*) args);

    /*
       setup space
     */
    x       = Calloc(N, double);
    c       = Calloc(N, double);
    b       = Calloc(N, double);
    y       = Calloc(N, double);
    samples = Calloc(N*NSAMPLES, double);

    /*
       read data
     */
    noise_variance = SQR(0.25);
    fp = fopen(DATAFILE, "r");
```

```
    for(i=0;i<N;i++) fscanf(fp, "%lf", &y[i]);
    fclose(fp);

    /*
       setup the posterior for x|y
     */
    for(i=0;i<N;i++)
    {
        b[i] = y[i]/noise_variance;
        c[i] = 1.0/noise_variance;
    }

    /*
       compute the conditonal mean and NSAMPLES samples
     */

    GMRFLib_init_problem(
        &problem, x, b, c, NULL,
        waproblem->graph, waproblem->Qfunc, waproblem->Qfunc_arg,
        NULL, NULL, GMRFLib_NEW_PROBLEM);
    for(k=0;k<NSAMPLES;k++)
    {
        GMRFLib_sample(problem);
        memcpy(&samples[k*N], problem->sample, N*sizeof(double));
    }

    for(i=0;i<N;i++)
    {
        printf("%d %f %f", i, y[i], problem->sub_mean[i]);
        for(k=0;k<NSAMPLES;k++) printf(" %f", samples[k*N+i]);
        printf("\n");
    }

    GMRFLib_free_problem(problem);
    GMRFLib_free_wa_problem(waproblem);
    GMRFLib_free_graph(wagraph);
    FREE(x); FREE(c); FREE(b); FREE(y); FREE(samples);

    return 0;
}
```

The program provides the output in Figure 2, for order=1 and $h = 50$, and order=2 and $h = 100$.

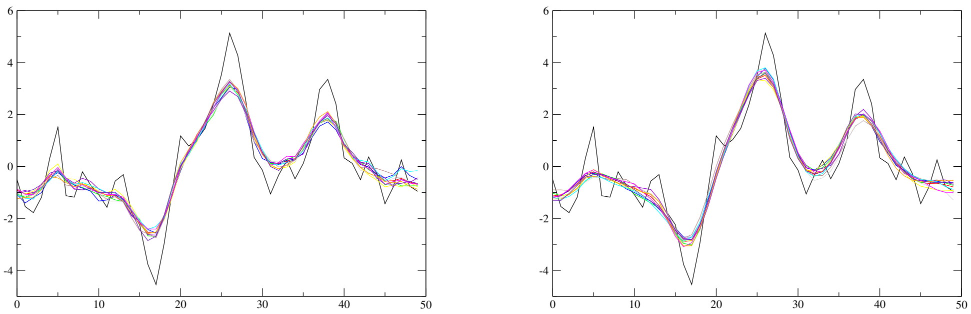

Figure 2: The observed and smoothed time-series, order=1 and h=50 to the left, and order=2 and h=100 to the right. The figures display the observed data, the posterior mean and 10 iid samples from the posterior.

5.4 A worked out example in spatial disease mapping, using block-updating

The spatial mapping of risk for a particular disease, based on the observed umber of incidences or the mortality in counties or other administrative zones, is of importance for formulation and validation of aetiological hypotheses. We will present a basic model used to describe such a problem, and how this can be implemented in `GMRFLib`.

Suppose that the area of interest is divided into n regions, as shown in Figure 3a. Let $y_1, \ldots, y_n$ be the number of deaths from the disease of interest in each of the n regions during the study period. A common assumption is that y_i is Poisson distributed with mean $E_i r_i$ where $\boldsymbol{E} = (E_1, \ldots, E_n)^T$ are known constants accounting for the number of inhabitants etc. The relative risks $\boldsymbol{r}$ are assumed to be of the form $\log r_i = x_i$, where $\boldsymbol{x}$ is an improper GMRF with density

$$\pi(x_1, \ldots, x_n \mid \kappa) \ \propto \kappa^{(n-1)/2} \exp\left(-\frac{\kappa}{2}\sum_{i\sim j}(x_i - x_j)^2\right). \tag{45}$$

The parameter κ is unknown and to be estimated, and is given a Gamma(a, b) prior. Hence, the conditional distribution of y_i given the GMRF $\boldsymbol{x}$ and the parameter κ is

$$y_i \mid x_1, \ldots, x_n, \kappa \ \sim \ \text{Poisson}(E_i \exp(x_i)) \tag{46}$$

We will now illustrate how the block-update routine in `GMRFLib`, `GMRFLib_blockupdate`, can be used to construct an efficient MCMC algorithm that updates $(\kappa, \boldsymbol{x})$ jointly. One joint update of $(\kappa, \boldsymbol{x})$ is generated by the following steps.

- First, a new value of κ, κ' is proposed by

  $$\kappa' = f\kappa \tag{47}$$

  where f has density

  $$\pi(f) \propto 1 + 1/f \tag{48}$$

  on the interval $[1/F, F]$, where $F > 1$. The role of F is as a ”step-length” for the proposal for κ. The reason for using (48) is that

  $$\frac{\pi(\kappa' \mid \kappa)}{\pi(\kappa \mid \kappa')} = 1. \tag{49}$$

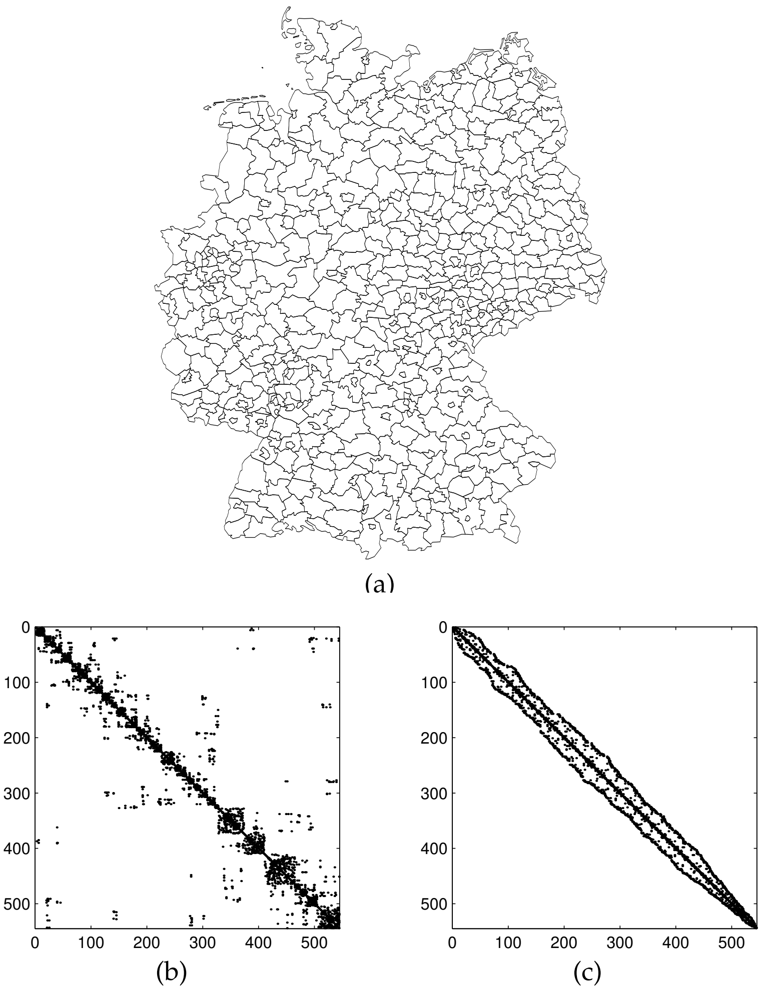

Figure 3: (a) shows the map of Germany with 544 regions, (b) shows the non-zero entries in the precision matrix $\boldsymbol{Q}$ for the GMRF using the numbering of the regions as in the definition of the map, (c) shows the same precision matrix after a permutation of the nodes to minimize the bandwidth.

- Next, conditionally on κ', a new value of the spatial field $\boldsymbol{x}$, denoted by $\boldsymbol{x}'$, is proposed from a GMRF approximation to the conditional posterior distribution of $\boldsymbol{x}$ given κ, given by

$$\pi(\boldsymbol{x}' \mid \kappa', \boldsymbol{y}, \boldsymbol{E}) \propto (\kappa')^{(n-1)/2} \exp\left(-\frac{\kappa'}{2}\sum_{i\sim j}(x'_i - x'_j)^2 + \sum_i y_i x'_i + \sum_i E_i \exp(x'_i)\right). \quad (50)$$

  This distribution is of the form (30), with $f(x_i;\theta) = \exp(x_i)$. The approximation is given by computing a second order approximation to $\exp(x'_i)$ around a point $\boldsymbol{x}^*$, which may depend on κ' and/or the current state $\boldsymbol{x}$. The default option is to let $\boldsymbol{x}^* = \boldsymbol{x}^*(\kappa')$ be the mode of $\pi(\boldsymbol{x}' \mid \kappa', \boldsymbol{y}, \boldsymbol{E})$. Another option is to expand around the current state, such that $\boldsymbol{x}^* = \boldsymbol{x}^*(\boldsymbol{x}, \kappa)$.

- Finally, the acceptance rate is computed, and the joint proposal accepted or rejected. Let

$$\widetilde{\pi}(\boldsymbol{x}' \mid \kappa', \boldsymbol{y}, \boldsymbol{E}, \boldsymbol{x}^*(\kappa', \boldsymbol{x})) \quad (51)$$

  denote the GMRF approximation. Then, the acceptance probability is $\min\{1, R\}$, where

$$R = \frac{\pi(\boldsymbol{x}', \kappa' \mid \boldsymbol{y}, \boldsymbol{E})}{\pi(\boldsymbol{x}, \kappa \mid \boldsymbol{y}, \boldsymbol{E})} \times \frac{\widetilde{\pi}(\boldsymbol{x} \mid \kappa', \boldsymbol{y}, \boldsymbol{E}, \boldsymbol{x}^*(\kappa', \boldsymbol{x}'))}{\widetilde{\pi}(\boldsymbol{x}' \mid \kappa, \boldsymbol{y}, \boldsymbol{E}, \boldsymbol{x}^*(\kappa, \boldsymbol{x}'))}. \quad (52)$$

  The block-update routine returns the log of the contribution to the acceptance probability from the GMRF proposal. However, this is based on the unnormalized posterior distribution, and the contributions from the normalizing constant and from the κ-proposal has to be added explicitly. Observe how this is done in the example program code below.

The efficiency of the sampling algorithm relies on a re-ordering of the graph such that the bandwidth corresponding precision matrix $\boldsymbol{Q}$ is small. In Figure 3 the effect of this re-ordering on the precision matrix is illustrated.

One refinement of the procedure described above is to constrain $\boldsymbol{x}$ to sum to 0. This seems reasonable, as the constants $\boldsymbol{E}$ usually account for the overall level. The constraint is easy to incorporate using `GMRFLib`, as illustrated by the example code shown below.

The following program, `example-blockupdate.c`, implements this scheme using the `GMRFLib` library. Note how the `char*` argument to various functions is used to transfer arguments.

The program runs on a 650MHz-laptop, doing about 11 iterations per second on the data in Figure 3, using $\boldsymbol{x}^*(\kappa)$ as the mode, and including the constraint that $\boldsymbol{x}$ should sum to 0.

Program code:

```
#include <assert.h>
#include <stddef.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#if !defined(__FreeBSD__)
#include <malloc.h>
#endif

/*
   include the require header-file
 */
#include "GMRFLib/GMRFLib.h"
```

```
/*
   some useful macros
 */
#define Calloc(n,type) (type *)calloc((size_t)(n),sizeof(type))
#define FREE(ptr) {if(ptr)free((void *)(ptr));ptr=NULL;}
#define MAX(a,b)  ((a) > (b) ? (a) : (b))
#define MIN(a,b)  ((a) < (b) ? (a) : (b))
#define SQR(x)    ((x)*(x))

/*
   the names of the graph and data
 */
#define GRAPH "germany.graph"
#define DATA  "germany.data"

/*
   parameters in the gamma-prior for kappa
 */
#define PRIOR_A 0.25
#define PRIOR_B 0.005

/*
   global variable G containg the data and the graph.
 */
typedef struct
{
    double *y;
    double *E;
    GMRFLib_graph_tp *graph;
}
Global;
static Global G;


double eu_gamma(double x, double a, double b)
{
    /*
       return the unnormalized log-density of a gamma-variable with mean a/b.
     */
    return (a-1.0)*log(x)-x*b;
}
int read_graph_and_data()
{
    /*
       read the graph and the data
     */
    int i;
    double dummy;
    FILE *fp;

    GMRFLib_read_graph(&G.graph, GRAPH);
    G.y = Calloc(G.graph->n, double);
    G.E = Calloc(G.graph->n, double);
    fp = fopen(DATA, "r");
    for(i=0;i<G.graph->n;i++)
        if (fscanf(fp, "%lf %lf %lf\n", &G.y[i], &G.E[i], &dummy) != 3) abort();
    fclose(fp);
    return 0;
}
int loglik(double *logll, double *x, int m, int idx, double *x_vec, char *arg)
{
    /*
       return the log-likelihood for a poisson, y_idx|... ~ Po(E_idx \exp(x_idx))
     */
    int i;
    double *y, *E;
```

```
    char **args;

    /*
       the data y and E come through the arg-pointer
     */
    args  = (char **)arg;
    y     = (double *)args[0];
    E     = (double *)args[1];

    /*
       compute the log-likelihood
     */
    for(i=0;i<m;i++) logll[i] = y[idx]*x[i]-E[idx]*exp(x[i]);

    return GMRFLib_SUCCESS;
}
double Qfunc(int node, int nnode, char *arg)
{
    /*
       return Q_ij, i=node, j=nnode, for the prior

       \pi(x) \sim \exp(-\frac{1}{2} \sum_{i\sim j} (x_i-x_j)^2)
     */

    GMRFLib_graph_tp *g;
    char **args;
    double kappa;

    args  = (char **)arg;                           /* the required variables come */
    g     = (GMRFLib_graph_tp *)args[0];            /* through the arg-pointer */
    kappa = *((double *)args[1]);

    return kappa*(node == nnode ? g->nnbs[node] : -1.0);
}
double scale_proposal(double x, double f, double *la)
{
    /*
       propose a new value, scale, on the interval [1/f, f], with
       density \propto 1+1/x.  this choice makes q(x,x')/q(x',x) = 1,
       in the acceptance ratio, when x' = scale*x.
     */
    double len = f - 1/f;
    if (la) *la = 0.0;
    if (f == 1.0) return x;
    if ((*GMRFLib_uniform)() < len/(len+2*log(f)))
        return (1/f + len*(*GMRFLib_uniform)())*x;
    else
        return pow(f, 2.0*(*GMRFLib_uniform)()-1.0)*x;
}
double gettime()
{
    /*
       simple routine for monitoring the cpu usage
     */
    return ((double)clock())/CLOCKS_PER_SEC;
}
int main(int argc, char **argv)
{
    int seed, i, counter=0;
    double fac, *x, *xnew, *d, kappa, kappa_new, lacc, eprob=0.0, timeref;
    char *arguments_new[2], *arguments_old[2], *arguments_loglik[2];
    GMRFLib_constr_tp *constr = NULL;
    GMRFLib_blockupdate_param_tp *blockupdate_param;
    GMRFLib_optimize_param_tp *optimize_param;

    printf("Usage: %s SEED\n", *argv);
    assert(argc > 1);
```

```
/*
   get the seed and init the random generator in GMRFLib
 */
seed = atoi(argv[1]);
printf("seed %d\n", seed);
(*GMRFLib_uniform_init)(seed);


/*
   get graph and data
 */
read_graph_and_data();

/*
   setup space
 */
x    = Calloc(G.graph->n, double);
xnew = Calloc(G.graph->n, double);
d    = Calloc(G.graph->n, double);

/*
   inital values
 */
kappa =  10.;
for(i=0;i<G.graph->n;i++)
{
    x[i] = MIN(2., G.y[i]/G.E[i]);
    d[i] = 1.0;
}

/*
   the scale-factor for proposing a new value for kappa
 */
fac = 1.5;

/*
   setup arguments to the log-likelihood function, which needs
   data (y and E), the graph, and kappa.
 */
arguments_loglik[0] = (char *)G.y;
arguments_loglik[1] = (char *)G.E;
arguments_new[0]    = (char *)G.graph;
arguments_new[1]    = (char *)&kappa_new;
arguments_old[0]    = (char *)G.graph;
arguments_old[1]    = (char *)&kappa;

/*
   choose whether or not the constraint \sum x_i = 0  is required
*/
if (1)
{
    GMRFLib_create_constr(&constr);
    constr->nc = 1;
    constr->a_matrix = Calloc(G.graph->n, double);
    for(i=0;i<G.graph->n;i++) constr->a_matrix[i] = 1.0;
    constr->e_vector = Calloc(1, double);
    constr->e_vector[0] = 0.0;
    GMRFLib_prepare_constr(constr);
}
else
    constr = NULL;

/*
   get default options for the blockupdate-routine and the
   optimizing-routine which is (optionally) used in the
   blockupdate-routine. now it's easy to change options if required.
```

```
     */
    GMRFLib_default_blockupdate_param(&blockupdate_param);
    GMRFLib_default_optimize_param(&optimize_param);

    /*
       start sampling, just go on forever
     */
    timeref= gettime();
    while(++counter)
    {
        /*
           propose a new value for kappa
         */
        kappa_new = scale_proposal(kappa, fac, NULL);

        /*
           conditional on kappa_new, propose a new value for the GMRF x
         */
        GMRFLib_blockupdate(
            &lacc,
            xnew, x,
            NULL, NULL,           /* zero vectors */
            NULL, NULL,           /* zero vectors */
            NULL, NULL,           /* zero vectors */
            d, d,
            loglik, (char *)arguments_loglik,
            loglik, (char *)arguments_loglik,
            NULL,                 /* none fixed values, update all */
            G.graph,
            Qfunc, (char *)arguments_new,
            Qfunc, (char *)arguments_old,
            NULL, NULL,           /* default values used here */
            NULL, NULL,           /* default values used here */
            constr, constr,
            optimize_param, blockupdate_param);

        /*
           the blockupdate-routine does not compute the normalization
           constant terms for the GMRF, nor the prior for kappa
         */
        lacc += (G.graph->n-1.)/2.*log(kappa_new/kappa)
            + eu_gamma(kappa_new, PRIOR_A, PRIOR_B)
            - eu_gamma(kappa, PRIOR_A, PRIOR_B);

        /*
           standard accept/reject
         */
        if ((*GMRFLib_uniform)()<exp(MIN(0.0, lacc)))
        {
            kappa = kappa_new;
            memcpy(x, xnew, G.graph->n*sizeof(double));
        }

        /*
           that's it !
         */
        eprob += exp(MIN(0.0, lacc));
        printf("lacc= %.4f  E(accept_prob)= %.4f iter/src= %.4f  kappa= %.4f\n",
               lacc, eprob/counter, 1./((gettime()-timeref)/counter), kappa);
    }
    return 0;
}
```

6 Installing and compiling with the `GMRFLib`-library

Download the latest version of the `GMRFLib`-library, at

```
http://www.math.ntnu.no/~hrue/GMRFLib/GMRFLib-latest.tar
```

Unpack `GMRFLib-latest.tar` by

```
tar xvf GMRFLib-latest.tar
```

and the files will unpack in directory `GMRFLib`.
Edit the `Makefile`, type

```
make
```

and install the library by

```
make install
```

and you may uninstall it later, if needed, by

```
make uninstall
```

The `Makefile` has some options in the top of the file, which are the defaults suitable for my Linux-laptop.

```
# USER OPTIONS
#
# where to install the library and headerfiles
PREFIX = /usr/local

# unncomment these two lines if you want the required Lapack and Blas routines
# to be compiled into the library.
#include .GMRFLib-Lapack-Blas-selection
#LBSEL=$(GMRFLib-Lapack-Blas-selection)

# select compilers and optimized compiler-options
CC = gcc
FC = g77
FLAGS= -O3 -funroll-all-loops -fno-f2c
```

Your need to decide about three matters.

1. The PREFIX is where to install the files, and the default choice is suitable for a system installation. If you want to install them locally, you might use `PREFIX=$(HOME)/local`.

   `GMRFLib` will install the library in `$(PREFIX)/lib`, the headerfiles in `$(PREFIX)/include/GMRFLib`, and the documentation in `$(PREFIX)/doc/GMRFLib`.

2. The `Lapack` and `Blas` libraries are often present by default on your computer, otherwise, `GMRFLib` can provide the routines needed, or you may obtain the full libraries from `http://www.netlib.org/lapack`. If you uncomment, by removing the "#"-character, the two lines in `Makefile` concerning these libraries, then all `Lapack` and `Blas` files needed for `GMRFLib` will be compiled into the `GMRFLib`-library.

3. Choose which compiler to use and the compiler options. Make sure to use compile options allowing “near optimal” performance, for example on a Sun Ultra, your might chose

   ```
   CC = cc
   FC = f77
   FLAGS= -fast
   ```

   and then recall to link your programs with the `-fast` option as well.

To compile your program using the `GMRFLib`-library, you may use (assuming `blas` and `lapack`-library is available on your computer)

```
   $(CC) $(FLAGS) -c -I$(PREFIX)/include my-prog.c
   $(FC) $(FLAGS) -o my-prog my-prog.o -L$(PREFIX)/lib -lGMRFLib -lblas
-llapack
```

Here, `$(CC)` is the `C`-compiler as defined in `Makefile` and similar with the other symbols. It is often easiest to link with the Fortran-compiler to combine both `C` and `Fortran` code. Sometimes you might change the last command to

```
   $(FC) $(FLAGS) -o my-prog my-prog.o \
         -L$(PREFIX)/lib -R$(PREFIX)/lib -lGMRFLib -lblas -llapack
```

If you install the `GMRFLib`-library in a common location, you may only need

```
   $(CC) $(FLAGS) -c my-prog.c
   $(FC) $(FLAGS) -o my-prog my-prog.o -lGMRFLib -lblas -llapack
```

A Default error handling function

This is the definition of the default error handling function `GMRFLib_error_handler`, which can be revised and replaced by a user-defined function, if preferred.

```
int GMRFLib_error_handler(const char *reason, const char *file, const char *function,
                          int line, const char *id, int errorno, const char *condition)
{
    /*
       this is the default error-handler
     */
    fprintf(stderr, "\n\n\tGMRFLib has recived error no [%1d]\n", errorno);
    fprintf(stderr, "\tReason    : %s\n", (reason ? reason : "(unavailable)"));
    if (condition) fprintf(stderr, "\tCaused by : if (!(%s)) error_handler()\n", condition);
    fprintf(stderr, "\tFunction  : %s\n", (function ? function : "(unavailable)"));
    fprintf(stderr, "\tFile      : %s\n", file);
    fprintf(stderr, "\tLine      : %1d\n", line);
    fprintf(stderr, "\tRCSId     : %s\n", id);
    fprintf(stderr, "\n");

    if (errorno != GMRFLib_SUCCESS) abort();      /* no reason to abort if ok */

    return GMRFLib_SUCCESS;
}
```

References

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J., Greenbaum, A., Hammerling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. (1995). *LAPACK Users' Guide, 2 edn*. Philadelphia: Society for Industrial and Applied Mathematics.

Cressie, N. (1993). *Statistics for Spatial Data*. Wiley, New York.

Gilks, W. R., Richardson, S., and Spiegelhalter, D. J. (1996). *Markov Chain Monte Carlo in practice*. Chapman & Hall, London, UK.

Golub, G. H. and van Loan, C. F. (1996). *Matrix Computations, 3 edn*. Johns Hopkins University Press, Baltimore.

Knorr-Held, L. and Rue, H. (2002). On block updating in Markov random field models for disease mapping. *Scandinavian Journal of Statistics*, xx(xx):xx–xx. (To appear).

Robert, C. P. and Casella, G. (1999). *Monte Carlo statistical methods*. Springer-Verlag, New York.

Rue, H. (2001). Fast sampling of Gaussian Markov random fields. *Journal of the Royal Statistical Society, Series B*, 63(2):325–338.

Rue, H. and Tjelmeland, H. (2002). Fitting Gaussian Markov random fields to Gaussian fields. *Scandinavian Journal of Statistics*, xx(xx):xx–xx. (To appear).