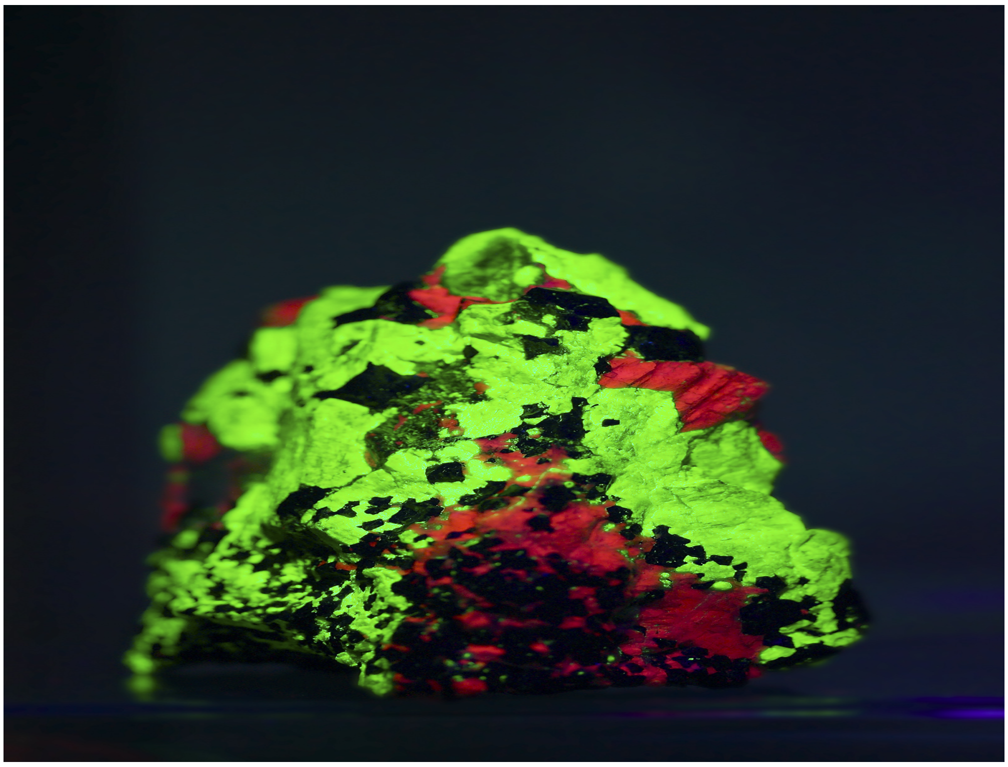
Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

An Implementation of Markov Random Fields

Overview and user's manual for version 0.1

Note


© www.photos.com 2006

Note no	SAND/06/06
Authors	Heidi Kjønsberg
	Harald H. Soleng
Date	17th October 2006

Norsk Regnesentral
Norwegian Computing Center
Postboks 114, Blindern
NO-0314 Oslo, Norway

Besøksadresse
Office address
Gaustadalléen 23
NO-0373 Oslo, Norway

Telefon · telephone
(+47) 22 85 25 00
Telefaks · telefax
(+47) 22 69 76 60

Internett · internet
www.nr.no
E-post · e-mail
nr@nr.no

Title	**An Implementation of Markov Random Fields**
Authors	**Heidi Kjønsberg, Harald H. Soleng**
Date	17th October 2006
Publication number	SAND/06/06

Abstract

The main purpose of these notes is to provide documentation of version 0.1 of the Markov random field implementation done for the Multipoint project. The report is mainly meant for internal use in the project, serving as a user manual for running the implemented program and as a guide for programmers wanting to modify it.

Keywords	Markov random field, implementation
Target group	Multipoint project participants
Availability	Internal
Project	Multipoint
Project number	808002
Research field	Reservoir characterization
Number of pages	30
© Copyright	Norwegian Computing Center

Contents

1 Introduction

The project of which this report is a part, aims at comparing and further develop different methods for multipoint statistics. Multipoint methods have been introduced as a tool for modelling geological structures, and they seem to provide better representations than classical variogram methods. Markov random fields constitute one type of multipoint methods.

To a large extent a comparison between Markov random fields and other multipoint methods has to be based on simulations. In particular this is true when comparing how well the different methods can reproduce patterns and essential characteristics of a given training image. It is therefore of great importance to establish a suitable software implementation of MRFs. We have implemented a program that uses traditional Markov chain Monte Carlo (MCMC) methods for simulating Markov random fields. Sequential simulation has been implemented as an optional choice, although the implementation for this has not yet been developed as far as the MCMC implementation. The implementation is done in C++.

2 MRF: the implemented models

This section provides the definition of a Markov random field (MRF) as well as a basic expression for its joint probability distribution. The section also presents the probability distributions that are supported by our implementation.

2.1 Markov random field definition

Consider a finite, regular grid in arbitrary many spatial dimensions, and let the one-dimensional index i label the cells of the grid. The set of all cells is $S = \{1, 2, \ldots, N\}$. In order to define a Markov random field it is necessary to introduce the concept of a *neighbourhood system*. Let ∂_i be a collection of cells, where the label i points to one specific cell. By definition, the set $\partial = \{\partial_i, i \in S\}$ is a neighbourhood system if

$$i \notin \partial_i, \text{ (not neighbour to oneself)} \tag{1}$$

and

$$j \in \partial_i \Leftrightarrow i \in \partial_j, \text{ (symmetry).} \tag{2}$$

The set of cells ∂_i is then the neighbourhood of cell i.

A Markov random field on the grid is specified with respect to the neighbourhood system ∂. By definition, a probability distribution $p(z_1, z_2, \ldots, z_N)$ that satisfies

$$p(z_i|z_{-i}) = p(z_i|z_j : j \in \partial_i), \tag{3}$$

is a Markov random field, provided that it also fulfills the positivity assumption

$$\forall i \; p(z_i) > 0 \Rightarrow p(z_1, z_2, \ldots, z_N) > 0. \tag{4}$$

The positivity assumption says that all combinations of single-cell configurations are valid. Equation (3) expresses that the conditional distribution $p(z_i|z_{-i})$ depends only on the cells in the neighbourhood of cell i. In this sense the interaction between the cells is limited by the neighbourhood system.

There exists a wide range of different neighbourhood systems. Some common examples, implicitely assuming the metric to be Euclidean, are

- 1st order neighbourhood: only grid cells that are nearest neighbours to cell i belong to ∂_i,
- 2nd order neighbourhood: only grid cells that are nearest or next nearest neighbours to cell i belong to ∂_i,
- the whole grid: $\partial_i = S \setminus \{i\}$.

The last example shows that any probability distribution that fullfills the positivity assumption is a Markov random field. The typical application of Markov random field theory assumes that each neighbourhood is relatively small compared to the grid size.

2.2 General expression based on the Hammersley–Clifford

There exists a profound theorem that gives a general expression for the form of a Markov random field: the Hammersley–Clifford theorem. The theorem states that a Markov random field can be expressed as

$$p(z_1, z_2, \ldots, z_N)/p(0,0,\ldots,0) = \exp\Bigg\{ \sum_{1\le i\le N} z_i F_i(z_i) + \sum_{1\le i<j\le N} z_i z_j F_{i,j}(z_i, z_j) + \sum_{1\le i<j<k\le N} z_i z_j z_k F_{i,j,k}(z_i, z_j, z_k) + \ldots + z_1 z_2 \ldots z_N F_{1,2,\ldots,N}(z_1, z_2, \ldots, z_N) \Bigg\}, \quad (5)$$

where the functions $F_{i,j,..,s} \neq 0$ only if *all* the cells $i, j, .., s$ are mutually neighbours. In particular, unless all cells in the whole grid are neighbours ($\partial_i = S \setminus \{i\},\ \forall i$), the function $F_{1,2,\ldots,N}$ will be zero.

Any set of cells where all set members are neighbours is called a *clique*. Hence the Hammersley–Clifford theorem expresses the Markov random field as a series expansion over 1-cliques, 2-cliques, up to and including the largest clique provided by the given neighbourhood system. This shows that any Markov random field is a Gibbs field, as the Gibbs field is defined in terms of a clique expansion of this kind. The contrary is also true, any Gibbs field is a Markov random field. That is, Markov random fields and Gibbs fields are the same thing.

The functions $F_{i,j,\ldots,s}$ are otherwise arbitrary in the sense that the corresponding p, defined through Eq. (5), will be a Markov random field for any choice of F-functions, provided they fullfill the "clique condition". In particular, a function F may be zero even when the corresponding cells provide a clique.

2.3 Implemented MRF models

Our implementation is done under the assumption of stationarity. That is, we assume the conditional probability distribution $p(z_i|z_j\,:\, j \in \partial_i)$ to be independent of the actual location of cell i, only depending on the facies configuration of cell i's neighbouring cells. Under this assumption we have implemented

1. General MRF: The form of the conditional probability function is given by

$$p(z_i|z_j\,:\, j \in \partial_i) \propto \exp\left\{ F_c(z_i) + \sum_l F_l(z_i, z_{i+l}) + \sum_l \sum_k F_{l,k}(z_i, z_{i+l}, z_{i+k}) + \ldots \right\}. \quad (6)$$

   Notice that the external field F_c is independent of the position i, and only the relative positions of the cells are used to label the interaction parameters. This is a consequence of the stationarity assumption.

2. General two-particle interactions: In this case all interactions of order higher than 2 are assumed to be zero. The conditional probability distribution thus has the general form

$$p(z_i | z_j : j \in \partial_i) \propto \exp \left\{ F_c(z_i) + \sum_l F_l(z_i, z_{i+l}) \right\}. \tag{7}$$

3. Pott's model, simple and generalized: This is a special form of the two-particle interaction model, where the interactions of Eq. (7) are assumed to be of the form

$$F_l(z_i, z_{i+l}) = \beta \, \delta_{z_i z_{i+l}} \, f_l. \tag{8}$$

   Here $\delta_{z_i z_{i+l}}$ is the Kronecker delta function, β is a parameter that gives the strength of the non-spatial factor of the interaction, and f is a function that determines spatial dependence. That is, if the two cells have the same facies, their interaction energy is β times a spatial function that depends on their relative position. If the two cells have different facies their interaction energy is zero. We refere to the special case of $f = 1$ as the simple Pott's model, and $f \neq 1$ as the generalized Pott's model.

The most general model above (number 1 in the list) does of course include the special cases listed as second and third. The reason why we have implemented the special cases of general two-particle interactions and Pott's model (simple and generalized) separately is partly due to the historic development of the program, partly to speed up the simulations whenever possible. Input parameters to the simulation program determines which model to consider. This is further explained in Section 3.

3 Program structure

This section describes the implementation of the Markov random field. The implementation is done in C++, with extensive use of the in-house proprietary library NRlib [1].

3.1 Environment

A model file provides input parameters to the simulation, with the model file being in xml-format. The model file specifies all necessary parameters, except the exact form of the Markov random field potential which in most cases is specified through its own input file, see Sections 4.6 and 5 for more details on this. It is not necessary to recompile the program in order to run a different interaction model or use different parameters. Simply provide a new model file and/or a new file for the input potential.

Appendix A gives an example of a model file and appendix B explaines how the program parses the file.

3.2 Overall structure

The main function in MRF is found in the `MRF.cpp`. Classes from NRlib are used to

1. run a stop-watch for timing purposes.
2. hold program data such as version number, copyright holder and so on.
3. read and keep command-line options.
4. read model file name from command line.
5. initialize the task manager with tailor-made actions.
6. parse the model file.
7. execute the model.

8. catch exceptions and report error messages.
9. report time usage and finish.

A flowchart for this function is shown in Fig. 1.

3.3 Actions

The implementation is designed so that it it possible to define different actions to be taken by the program. So far the only action that is fully implemented is simulation. One more action, allowing for the calculation of likelihood and different probability functions for a given input image, is presently being developed, but will not be described here.

Global parameters valid for all actions are read and kept by **GlobalCommands**, declared and defined in `globalcommands.h` and `globalcommands.cpp`. The first template argument given to the **NRlib::UI::TaskManager** initializer function defines the global commands. Subsequent actions can be declared by adding new template arguments in this function. An action class, e.g., **FaciesSimulation** should inherit from **NRlib::UI::Action** and should contain a copy constructor, a Parse and an Execute function.

4 Input parameters

The input parameters for simulations are described in the following, grouping parameters that naturally belong together.

4.1 Overall simulation specification

This section describes the parameters that determine which simulation method to use, the number of iterations, the initial grid configuration, and the number of independent realizations to be made.

- <method>: Two types of simulation methods have been implemented, Markov Chain Monte Carlo simulations (MCMC) and Sequential Simulations.
  Markov Chain Monte Carlo simulations: The main implementation uses standard MCMC simulations using a single site Gibb's sampler. The simulation starts with some (arbitrary) grid configuration, then updates each cell a number of times. Each update is done using the conditional probability $p(z_i|z_{\partial_i})$. Theoretically, as the number of updates increases, the probability distribution for the grid configurations converges towards the joint probability $p(z_1, z_2, \dots, z_N)$.

  Sequential simulations: Sequential simulations are to some extent supported by the implemented program. For this kind of simulation the initial grid is empty, and the cells are sequentially assigned a facies throughout the simulation. Each cell in the grid is visited only once, and the assignment of facies is based on the conditional probability $p(z_i|z_{\eta_i})$, where η_i is the so-called sequential neighbourhood of cell i. Our implementation allows the user to specify η_i, see Section 4.5 for further details. The probability $p(z_i|z_{\eta_i})$ is assumed to have a form similar to the two-particle version of Section 2.3. This is further explained in Section 4.6.

- <iterations>: The number of iterations per cell. Will be reset to 1 if sequential simulation is chosen

- <initialize><type>: The parameter <type> determines the start configuration of the grid in the case of MCMC simulations. If specified to 'random' a random initial configuration is

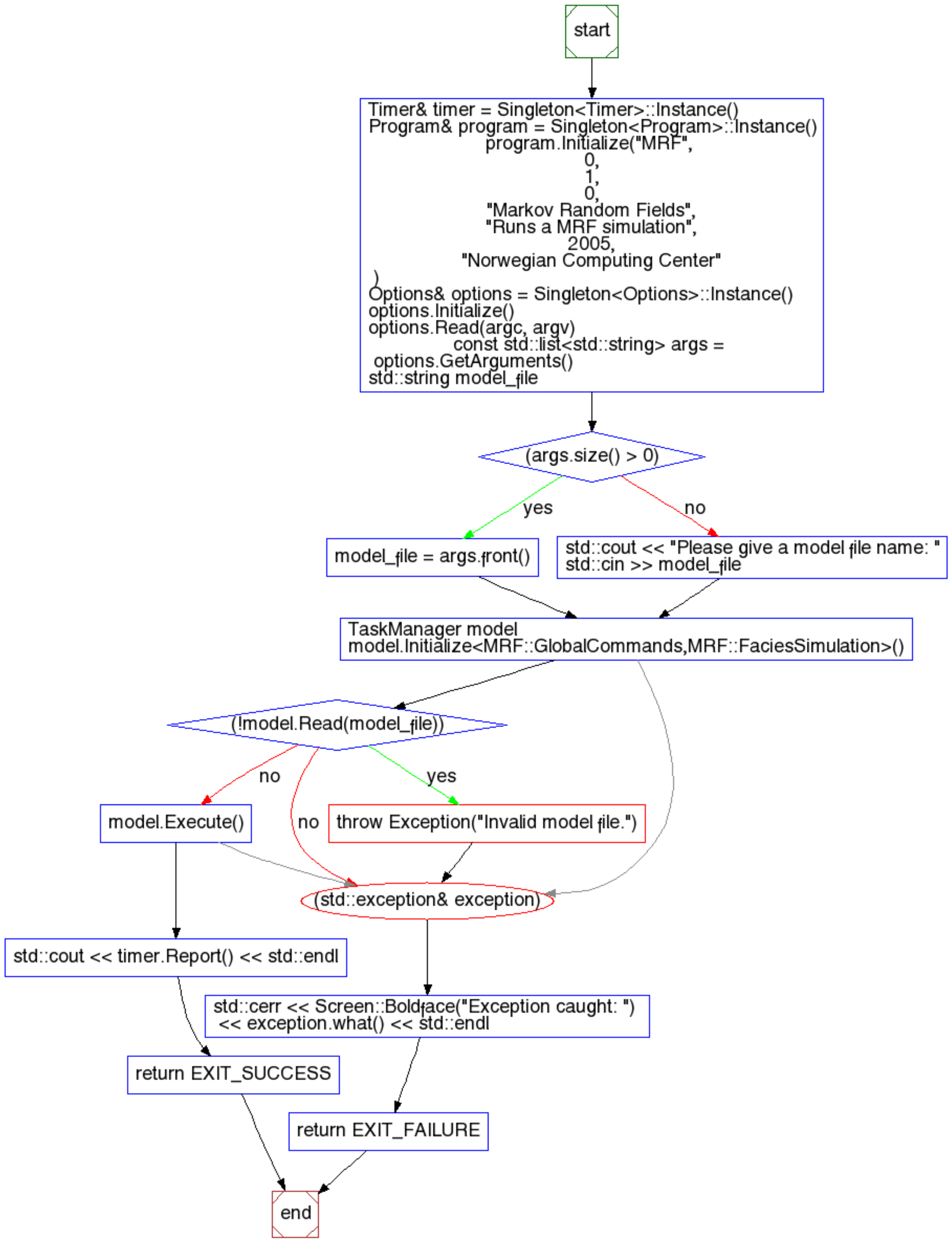

start
Timer& timer = Singleton<Timer>::Instance()
Program& program = Singleton<Program>::Instance()
program.Initialize("MRF",
0,
1,
0,
"Markov Random Fields",
"Runs a MRF simulation",
2005,
"Norwegian Computing Center"
)
Options& options = Singleton<Options>::Instance()
options.Initialize()
options.Read(argc, argv)
const std::list<std::string> args =
options.GetArguments()
std::string model_file
(args.size() > 0)
yes
no
model_file = args.front()
std::cout << "Please give a model file name: "
std::cin >> model_file
TaskManager model
model.Initialize<MRF::GlobalCommands,MRF::FaciesSimulation>()
(!model.Read(model_file))
no
no
yes
model.Execute()
throw Exception("Invalid model file.")
(std::exception& exception)
std::cout << timer.Report() << std::endl
std::cerr << Screen::Boldface("Exception caught: ")
<< exception.what() << std::endl
return EXIT_SUCCESS
return EXIT_FAILURE
end

Figure 1. Flowchart of **main** in `MRF.cpp`.

generated by the progam, if set to 'constant' all cells have $z_i = 0$ in the initial configuration. For sequential simulations the simulation starts with an empty grid, and the parameter has no effect

· <realizations>: The desired number of independent realizations should be specified. Each independent realization is achieved by starting from an independent initial configuration and then run the specified number of iterations for each cell on the grid. For sequential simulation the number of iterations is 1.

Parameters in model file	**Description/Details**	**Valid values**
<method>mcmc</method>	Specifies whether MCMC or sequential simulation is to be used	'mcmc' or 'sequential'
<iterations> 1000 </iterations>	The number of iterations per cell. Will be reset to 1 if sequential simulation is chosen	1,2,3,...
<initialize> <type>random</type> </initialize>	<type>: Specifies the initial grid configuration for MCMC simulations	'random', 'constant'
<realizations> 1 </realizations>	The desired number of independent realizations	1,2,...

4.2 Path

The path specifies in which order the grid cells are visited. It can be sequential (cell number $1, 2, 3, \ldots$) or random. If a random path is chosen, the MCMC simulation will use a reshuffled path for each grid update. For sequential simulations each cell is visited only once, using the specified path to run through the grid.

Parameters in model file	**Description/Details**	**Valid values**
<path>sequential</path>	Specifies the path to be used	'sequential', 'random'

4.3 Annealing

Annealing can be used in the MCMC simulations. It slowly brings the system down from a starting temperature T_{max} to the default temperature $T_\infty = 1$. The temperature develops in steps according to the formula $T_{k+1} = \epsilon(T_k - 1) + 1$. Three parameters must be specified:

· T_{max}: starting temperature.

· n_T: the number of grid updates per temperature.

· ϵ: determines how much the temperature is decreased in each step, see the formula above.

Parameters in model file	**Valid values**
<annealing> <max-temperature>10</max-temperature> <temperature-reduction-factor>0.9 </temperature-reduction-factor> <steps>10</steps> </annealing>	<max-temperature>: non-negative real number. Zero means no annealing. <temperature-reduction-factor>: real number in $\langle 0, 1 \rangle$ for annealing schedule <steps>: positive integer (grid updates per temperature update)

Annealing is useful mainly for MCMC simulations. If the simulation method is set to 'sequential' and annealing is nevertheless used, then this simulation will be performed at the temperature T_{max}.

4.4 Grid size

The grid is assumed to be cell-centered Cartesian. The parameters to be specified describe grid size in x, y, and z-direction.

Parameters in model file	Description/Details	Valid values
<grid> <nx>100</nx> <ny>100</ny> <nz>1 </nz> </grid>	Grid size, the number of cells in x, y, and z-direction	<nx> = 1,2,3,...

4.5 Neighbourhood

In a Markov random field simulation setting it may be convenient to think about the doublet (neighbourhood, interaction potential) in two slightly different ways. In the first of these, the interaction potential and neighbourhood are defined independently. Think of this as a situation where there is a physical system with interactions, strong or weak, at arbitrary inter-particle lags. Superpositioned on this we impose a neighbourhood structure. The resulting MRF model consists of all interactions for which the interacting particles are within one anothers neighbourhood. All other interactions are set to zero in the MRF model. This way of thinking about the doublet is convenient for exploring how a change in the neighbourhood structure affects the results of the MRF simulation, provided the underlying interactions are fixed. One example is to study the effect of neglecting interactions at long lags in a two-particle interaction model.

The other way to think about the doublet is to let the neighbourhood be implicitely defined by the interaction potential. That is, if two particles interact via a non-zero potential, then they are per definition in one anothers neighbourhood.

These two ways of thinking about the doublet are of course completely interchangable in the sense that any model that is set up using one point of view can easily be set up using also the other point of view. Which viewpoint that is seen as the most natural or intuitive is merely a matter of taste, and has no effect on the simulated results.[1]

Our implementation of Markov random fields supports explicitely both these points of view, provided that the interactions are limited to two-particle interactions. For general interaction models, only the second point of view is supported. The different interaction models are described in Section 4.6. This section describes the parameters that are to be specified if the neighbourhood is defined independently of the interaction potential, i.e. adopting the first viewpoint described above.

Two neighbourhood types are implemented, specified by the parameter <type>. The parameter can take two values, 'box' or 'sphere'. For each of these there are additional parameters that need to be specified. They are described in the following.

· If neighbourhood <type> is 'box': Box neighbourhoods are supported, both for the MCMC simulation and for sequential simulation. Periodic boundary conditioned are used if the neighbourhood type is 'box'

1. The second method requires a new input potential file for each change of neighbourhood. Hence if the main purpose of the simulations is to study the effect of changing the neighbourhood, for an otherwise fixed potential, the first method may imply less work on producing input files.

  - Box size in x-, y-, and z-direction; <nx>, <ny>, <nz>
  - Shift of center in x-, y-, and z-direction; <shiftx>, <shifty>, <shiftz>. If shifts are set to 0, then the box is centered at the cell in question, with <nx> cells in both positive and negative x-direction, and similarly for <ny> and <nz>
  - For sequential simulations there is also a parameter <max> that allows the user to limit the number of last visited cells to be checked for whether they are part of the given box or not. That is, of all previously visited cells, each of the last <max> cells are considered being part of the sequential neighbourhood, provided they are inside the given box. This means the sequential neighbourhood depends on the path. Normally the parameter <max> should be set to a number larger than the number of cells in the grid, in which case all visited cells that are within the box are included in the sequential neighbourhood. Another extreme is to use a box larger than the grid and define the sequential neighbourhood to consist of all the last <max> visited cells, no matter where in the grid they are

· If neighbourhood <type> is 'sphere': Sperical neighbourhoods are supported for the MCMC simulation, but not for sequential simulation (an error message and program abortion results if this is attempted)

  - Radius of sphere, <radius>

The parameters of this section are neglected by the program if the interaction method is set to 'general'. In that case the input potential itself implicitly defines the neighbourhood.

Parameters in model file	**Description/Details**	**Valid values**
<neighbourhood>		
<type>box</type>	<type>: neighbourhood type	<type>: 'box', 'sphere'
<nx>10</nx> <ny>10</ny> <nz>0</nz>	<nx>, <shiftx>: box size and shift of center, x-direction	<nx>, <shiftx>: 0,1,2,...
<shiftx>0</shiftx> <shifty>0</shifty> <shiftz>0</shiftz>		
<radius>1</radius>	<radius>: radius of spherical neighbourhood	<radius>: 0,1,2,...
<max>3001</max>	<max>: the number of cells to be checked for being inside box if sequential simulation	<max>: 0,1,2,...
</neighbourhood>		

4.6 Interaction type

The interactions can for MCMC simulations be of four types:

· General interactions ('general'): In this case there is no restriction on the joint probability distribution other than that given by Eq. (6). An input file that specifies the potential *must* be given. The file specifies all interactions, and implicitely also the neighbourhood. The file format is described in Section 5.1. For this interaction type the neighbourhood parameters of Section 4.5 are ignored, even when/if they are explicitely listed in the model file.

- General two-particle interactions ('two-particle'): In this case the joint probability distribution is restricted to have the form given by Eq. (7). An input file that specifies the potential *must* be given. The file format is described in Section 5.2. All interactions listed in the input file will be used, *provided* that the interacting particles are within each other neighbourhood, with the neighbourhood being defined by the parameters of Section 4.5.
- Generalized Pott's model ('generalized potts'): This is a special form of the two-particle interaction model, and the interaction is given by Eq. (8). An input file that specifies the spatial part f of the interaction *must* be given, the format is described in Section 5.3. The neighbourhood is specified by the parameters in Section 4.5.
- Simple Pott's model ('potts'): The interaction is given by Eq. (8), but with $f = 1$. Any input file for interaction potential will be ignored. The interactions are analogous to those of the Ising model, but an Ising model not necessarily only for nearest neighbours. The neighbourhood is specified by the parameters in Section 4.5.

It is possible to use the interaction type 'general' also to simulate the Pott's models as well as more general two-particle interaction models; just specify the input file in accordance with the desired model.

Sequential simulation can be carried out for two-particle interaction models, i.e. for the parameter <type> set to 'two-particle', 'generalized potts' or 'potts'. The sequential conditional probability $p(z_i|\eta_i)$ is then assumed to have the form shown on the right hand side of Eq. (7), alternatively as specified by Eq. (8). Be aware that this does *not* imply that the probability distribution $p(z_i|z_{-i})$ is on the same form[2]. For sequential simulations the interaction type 'general' has not yet been implemented. If this is nevertheless attempted, an error message will be written to screen and to the simulation log, and the program will abort.

Parameters in model file	**Description/Details**	**Valid values**

<interaction>
    <type>potts</type>
    <beta>0.5</beta>
 	
    <input-file>
      filename.dat
    </input-file>
 	
</interaction>
	With <type>general</type> an input file must be provided, <beta> is ignored, and the neighbourhood is implicitly defined through an input file.
	
	With <type>two-particle</type> an input file must be provided, and <beta> is ignored.
	With <type>generalized potts</potts> an input file for spatial part must be provided, <beta> determines the strength of the facies dependency of the interaction
	
	With <type>potts</potts> any input file for spatial part will be ignored, <beta> determines the strength of the facies dependency of the interaction
	
	File formats are described in Section 5
		<type>: general, two-particle, generalized potts, or potts;
		<beta> $\in R$
		
		<input-file>: Any existing text file, specific format

2. The relationship between the sequential probability distribution and $p(z_i|z_{-i})$ is discussed in detail in the report [2].

The path of the input potential file should be absolute or relative to the folder <topdir> described in Section 4.9.

4.7 Lithology

The number of facies to be used in the simulation is communicated to the program by listing (arbitrary) facies names. The interpretation of these names in the program is that they correspond to the cell's facies indicator as $z_i = 0, 1, 2, \ldots$, respective to the order of the facies listing[3]. The program is so far used and tested only for binary facies values, i.e. two listed names, but is made so as to accept up to 256 of facies types.

For each facies the user of the program can specify a target value for the volume fraction of this facies. If using the fraction control option (explained shortly), the program will then automatically adjust its parameters so that the measured volume fractions converge towards the target values. Thus for each facies the following parameters should be specified:

· <name>: Facies name

· <fraction>: Target value of volume fraction, to be used in fraction control (see next paragraph)

The volume fraction of each facies can be controlled throughout the simulation. The algorithm used is equivalent to adjusting the external field of the model, the adjustment being based on the target value per facies in combination with measurements done throughout the simulations. The algorithm is adopted from another project at NR/Sand, where its utility has been tested. It is described in the following:

· <fraction-control>; the change parameter S: If we define a as $a = S(1 - f/f_0)$, where f is measured volume fraction and f_0 is target value, the conditional probability for this facies is changed with the factor e^{4a}. The value $S = 0$ means fraction control is not performed. Recommended value is 1.

Parameters in model file	Description/Details	Valid values

<lithology>
  <facies>
    <name>sand</name>
    <fraction>0.7</fraction>
  </facies>
  <facies>
    <name>shale</name>
    <fraction>0.3</fraction>
  </facies>
</lithology>
	Names and target global volume fractions.
	<name> and <fraction> must be specified for each facies.
		<name>: Any text
		Target values <fraction> should be $\in R+$, and must sum to 1

<fraction-control> 1 </fraction-control>	Fraction control. The parameter S that determines the change in external field, given that target value and measured volume fraction are unequal	$R+$. Recommended value is 1. Set to 0 if fraction control is not to be used

4.8 Output

Several output files are automatically generated by the simulation program. These have default names that may be changed if desired. In addition there are two parameters describing the frequency of logging.

3. For correct interpretation of simulation results the input potentials of Section 4.6 should reflect this ordering in the sense that $z_i = 0$ means the first facies listed, $z_i = 1$ the next, and so on.

· Output files: There are three kinds of log files

  – <measurements>: File where the instantaneous mean facies in the grid is periodically appended throughout the simulation. May be used for convergence checking. Default name: measurementlog.txt
  – <result>: File where the final realization of the grid is written when simulation is over. Default name: result.dat
  – <temp>: Files where the temporary grid configuration is periodically written. Each writing is done to a unique file that is automatically labelled with a number by the program. The number is prepended to the given file name. Default name: tmpresult.dat

· Log frequencies:

  – <writefreq>: The number of iterations between each measurement and writing of mean facies
  – <gridfreq>: The number of iterations between each writing of instantaneous grid configuration

Parameters in model file	**Valid values**

<output>
    <measurements>
      measurementlog.txt
    </measurements>
    <result format="sgems">result.dat</result>
    <temp format="sgems">tmpresult.dat</temp>
 	
    <gridfreq>500000</gridfreq>
    <writefreq>10000</writefreq>
</output>
	
	Any file names may be used, but keep the file extensions as they are here (.txt, .dat, .dat). Formats may be “sgems” or “storm”
	
	<gridfreq>, <writefreq> = 1,2,. . .

4.9 File directories

Two names specify to which folder the simulation results are stored. The parameters holding these two names are called <topdir> and <prefix>. The folder with the name given by <prefix> is a subfolder under the folder <topdir>. Both folders are created if they do not already exist.

Parameters in model file	**Valid values**
<topdir> topdir </topdir>	Any folder name
<prefix> sim-name </prefix>	Any folder name

4.10 Miscellaneous

Seed for the random number generator[4] may be provided. In addition there are two parameters that specify the level of detail with which simulation log file information and information to screen is given. Log file information is written to the file mrf.log.

· <seed>: The seed for the programs random number generator. If not specified, seed is computed based on the computer’s clock

· <logfile><level>: Information level to be used for writing to file

4. Note that the Mersenne twister implementation gives different random number sequences on 32 bit and 64 bit platforms.

· <screen><level>: Information level to be used for writing to screen

Parameters in model file	**Valid values**
<seed> 345 </seed>	Any positive integer or nothing

<messages>
  <logfile>
    <level>5</level>
    <name>logs/mrf.log</name>
  </logfile>
  <screen>
    <level>5</level>
  </screen>
</messages>
	Valid information levels:
	0 no messages given
	1 severe error messages
	2 warnings
	3 information messages
	4 detailed information
	5 debugging messages
	6 requires DEBUG compile flag

5 Format of input files

This section describes the format in which the potential functions should be listed in order for the program to interprete them correctly. The potential file lists the potential values in a specific order, and it can for instance be made by using MatLab. Whether the listed values result from evaluating some known mathematical function or from estimating parameters from a training image is up to the user.

5.1 General interactions

This format of the input file is to be used whenever the parameter specifying the interaction type is set to 'general', see Section 4.6.

In this case the interactions are only restricted by the Hammersley–Clifford theorem. A convenient expression for the conditional probability is then

$$p(z_i|z_{\partial_i}) \propto \exp\{\sum_{C:i\in C}\sum_{z_C}\theta_{z_C}\}. \tag{9}$$

The outer sum is over all cliques such that cell i is in the clique, the inner sum is over all configurations z_C of the clique C. The parameter θ_{z_C} gives the strength with which this configuration contributes to the overall potential. If $\theta_{z_C} = 0$ the configuration has no effect on the potential. Eq. (9) gives the key to interpreting the potential file that is described in the following. Only parameters $\theta_{z_C} \neq 0$ will be listed in the file for the obvious reason of saving memory.

5.1.1 File format

The potential file should start with 4 integers that give information about the neighbourhood and number of facies. The integers are, in order:

$$L_x, L_y, L_z, K. \tag{10}$$

The interpretation of these integers is that the size of the neighbourhood is $(2L_x+1)(2L_y+1)(2L_z+1)$, with a maximum clique size $(L_x+1)(L_y+1)(L_z+1)$. The number of facies is K. The four integers are used by the program to interprete the remaining lines of the input file in a consistent way.

All remaining lines of the input file should have the following format

$$I_{\text{new}} \quad \eta \quad N \quad \{\alpha_i\}_{i=1}^N \quad \{f_i\}_{i=1}^N \quad \theta$$

where

N is the number of cells in the considered clique,

$\{\alpha_i\}_{i=1}^N$ gives 1D indices that determine the locations of the clique cells, relative to an internal labelling convention of the maximum clique,

$\{f_i\}_{i=1}^N$ is the facies indicators of the cells, with each $f_i \in \{0, 1, \ldots, K-1\}$

θ is the parameter that gives this clique configuration's contribution to the potential, see Eq. (9).

The two integers I_{new} and η provide extra information that allows the program to be more efficient. Their interpretation is

$$I_{\textbf{new}} = \begin{cases} 1 & \text{if this row is the first row in a sequence of one or more rows} \\ 0 & \text{otherwise,} \end{cases}$$

η gives the number of subsequent rows with the same set $\{\alpha_i\}_{i=1}^N$.

There is no harm done by setting $I_{\text{new}} = 1$ and $\eta = 1$ in all lines, but the simulation will be somewhat slower.

5.1.2 Internal labeling of cells in maximal clique

The maximal clique has dimensions $(L_x+1)(L_y+1)(L_z+1)$. Figure 2 illustrates the labelling of cells, i.e. the possible values in the set $\{\alpha_i\}_{i=1}^N$, for a 2D case. The extention to 3D is similar. The 1D-index α is related to the 3D lags l_x, l_y, l_z by

$$\alpha = l_x + l_y(L_x+1) + l_z(L_x+1)(L_y+1), \tag{11}$$

where $l_x \in \{0, 1, \ldots, L_x\}$, and similarly for l_y and l_z.

5.1.3 Examples of potential file

In the following we give an example of the potential file on the format corresponding to general interactions. The example represents the 2D nearest neighbour Ising model with an interaction strength $\theta = 0.5$. The file should then look like this:

```
1 1 0 2
1 2 2 0 1 0 0 0.5
0 2 2 0 1 1 1 0.5
1 2 2 0 2 0 0 0.5
0 2 2 0 2 1 1 0.5
```

Figure 3 illustrates the clique configurations corresponding to this file. The figure shows from left to right lines 2-5 of the file. The numbers indicate the facies of the cells, and the cells with no number do not belong to the clique considered in each case . The interaction parameter θ is in each case indicated above the figure.

A slightly more complex example of potential file is given by

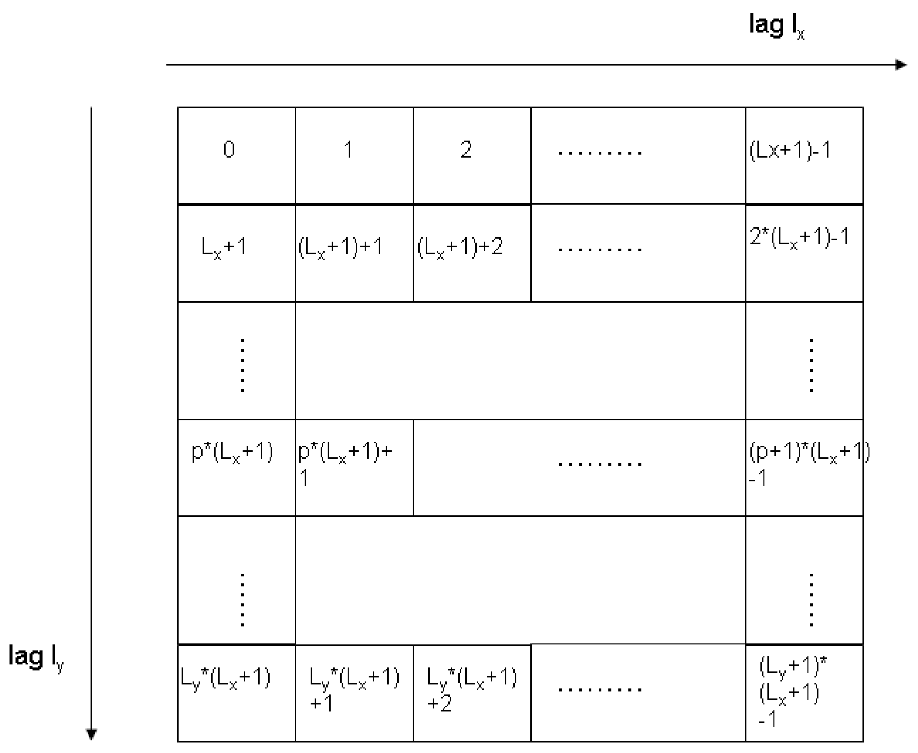


Figure 2. Convention used for indexing cells in max clique, 2D illustration

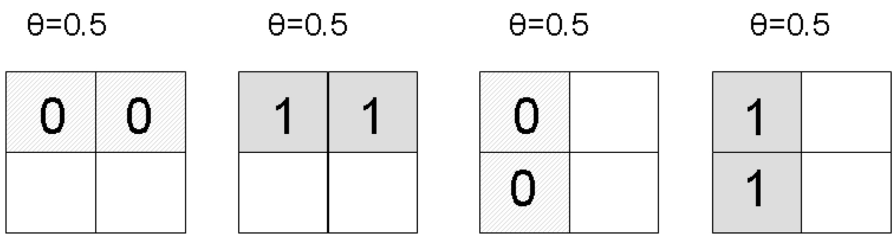


Figure 3. Clique configurations corresponding to the file example for the Ising model

```
2 1 0 2
1 4 2 0 1 0 0 0.2
0 4 2 0 1 1 1 -0.5
0 4 2 0 1 0 1 1.5
0 4 2 0 1 1 0 1.5
1 2 4 0 1 3 4 1 1 1 1 -0.3
0 2 4 0 1 3 4 0 0 1 1 1.2
1 3 6 0 1 2 3 4 5 1 0 1 0 1 0 0.1
0 3 6 0 1 2 3 4 5 0 1 0 1 0 1 0.4
0 3 6 0 1 2 3 4 5 1 0 1 1 0 1 -2.0
```

Figure 4 illustrates the clique configurations of this file. Be aware that this example only illustrates the file format and is not guaranteed to represent a stationary MRF model.

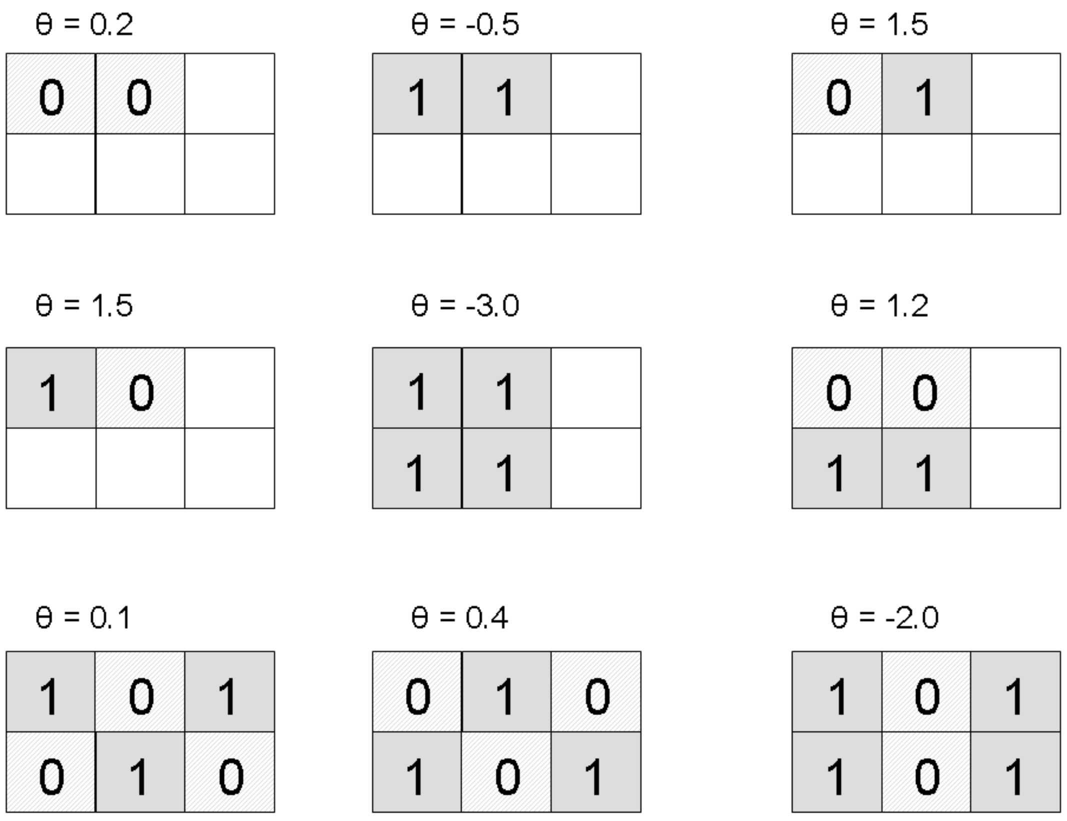


Figure 4. Clique configurations corresponding to the more general file example.

5.2 Two-particle interactions

This format of the input file is to be used whenever the parameter specifying the interaction type is set to 'two-particle' (see Section 4.6).

At the very beginning of the file the integers L_x, L_y, L_z, K must be given. These are used by the program to identify the index of the correct file element for a given relative position $\mathbf{\Delta r}_{ij}$ of the two interacting particles. The interpretation of these parameters are the same as for interaction type 'general', see Section 5.1, but the actual neighbourhood used to specify the MRF model may be different, see Section 4.5.

For general two-particle interactions both spatial dependency and dependency on facies must be listed. Recall from Eq. (7) that the general form of the probability distribution is assumed to be

$$p(z_i | z_j : j \in \partial_i) \propto \exp\{F_c(z_i) + \sum_l F_l(z_i, z_{i+l})\}. \tag{12}$$

The two-particle interactions $F_{i,j}(z_i, z_j)$ are assumed to be listed for a number of relative positions $\mathbf{\Delta r}_{ij} = (\Delta x_{ij}, \Delta y_{ij}, \Delta z_{ij})$ that constitute a box. That is, we assume

$$\Delta x_{ij} \in \{-L_x, -L_x + 1, \ldots, -1, 0, 1, \ldots, L_x - 1, L_x\}, \tag{13}$$

$$\Delta y_{ij} \in \{-L_y, -L_y + 1, \ldots, -1, 0, 1, \ldots, L_y - 1, L_y\}, \tag{14}$$

$$\Delta z_{ij} \in \{-L_z, -L_z + 1, \ldots, -1, 0, 1, \ldots, L_z - 1, L_z\}. \tag{15}$$

The positions are listed according to the rule: looping over x is done before looping over y is done before looping over z. The first distance for which a potential value is listed is $(\Delta x_{ij}, \Delta y_{ij}, \Delta z_{ij}) = (-L_x, -L_y, -L_z)$.

For each distance there is a list of K^2 elements, where K is the number of facies. Each of these elements corresponds to a two-particle facies configuration $(z_i, z_j) = (k_i, k_j)$. The order of these configurations is assumed to correspond to the nested loops

```
for k1 = 0:(K-1)
   for k2 = 0:(K-1)
      ‘‘list potential value G’’
   end
end
```

Combining the distance and facies conventions, the file should list the potential according to the order of the following nested loops

```
for z = -Lz:Lz
  for y = -Ly:Ly
    for x = -Lx:Lx
      for k1 = 0:(K-1)
        for k2 = 0:(K-1)
          ‘‘list potential value G’’
        end
      end
    end
  end
end
```

An external field can also be included in the input file. It should be listed as the potential value for $x = y = z = 0$ and $k_2 = k_1 = 1$ in the loop above.

This format of the input file is useful if the interaction potential is a known mathematical function. In this case the file is easily generated by using the nested for-loops described above. If the potential is not a known mathematical function it may be easier or more convenient to generate the file on the format used by the interaction mode ‘general’, i.e. the format described in Section 5.1. If so, the simulation itself must of course be run using the ‘general’ mode.

5.3 Generalized Pott’s interactions

If the interaction type is ‘generalized potts’ the expression for the interaction is, according to Eq. (8), written

$$F_l(z_i, z_{i+l}) = \beta\, \delta_{z_i z_{i+l}}\, f_l.$$

In this case the input file is only assumed to list the spatial dependency f_l. Apart from this restriction the convention for the order of the elements is analogous to what was said for the general two-particle interaction in the previous section. Hence the order in which the file should list the spatial potential is

```
for z = -Lz:Lz
  for y = -Ly:Ly
    for x = -Lx:Lx
        ‘‘list potential value f’’
    end
  end
end
```

In addition, at the very beginning of the file the three integers L_x, L_y, L_z must be given. These are used by the program to identify the index of the correct file element for a given lag l.

5.4 Closing remarks on interaction models

For a new user of the program it is probably most convenient to stick to the interaction model ‘general’ and the corresponding format of input potential file. This interaction model has the simplest file format, and as it can handle all cases, including the two-particle specific cases described earlier, it may be a useful attitude to concentrate on this kind of interaction model. We also expect that further extensions of the program will focus on using this model, and then the interaction models ‘two-particle’, ‘generalized potts’, and ‘potts’ will phase out.

6 Conclusion

Markov random fields have been implemented in a C++ framework called MRF. User input is taken from an Xml model file. In this report we have documented the overall structure of the program, the syntax and parsing of the model file with the help of NRlib. Hence, the document is both a user guide and a programmers guide.

The MRF software is a research tool under development, and features and specifications may change without notice.

References

[1] Harald H. Soleng. NRlib: A C++ toolbox. Note SAND/10/05, Norwegian Computing Center, Oslo, Norway, November 2005.

[2] Heidi Kjønsberg and Ingeborg Ligaarden. Second order markov mesh models described as markov random fields. Note SAND/07/06, Norwegian Computing Center, Oslo, Norway, 2006.

A Example of model file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/css" href="http://intern.nr.no/sand/css/cohiba.css"?>
<mrf version="0.1">
   <title>Example of MRF model file</title>
   <!-- Changed by: Harald H. Soleng, 25-Sep-2006 -->

   <name>Test model</name>

   <messages>
   <logfile>
     <level>5</level>
     <name>logs/mrf.log</name>
   </logfile>

   <screen>
     <level>5</level>
   </screen>
   </messages>

   <topdir>./</topdir>

   <prefix>real</prefix>

   <seed>345</seed>

   <action type="simulate facies">

     <method>mcmc</method>

     <annealing>
       <max-temperature>10</max-temperature>
       <temperature-reduction-factor>0.9</temperature-reduction-factor>
       <steps>10</steps>
     </annealing>
     <fraction-control>1</fraction-control>

     <iterations>1000</iterations>

     <realizations>1</realizations>

     <grid>
       <nx>100</nx>
       <ny>100</ny>
       <nz>1</nz>
     </grid>
     <lithology>
       <facies>
          <name>sand</name>
          <fraction>0.55</fraction>
       </facies>
       <facies>
         <fraction>0.45</fraction>
         <name>shale</name>
       </facies>
     </lithology>

     <initialize>
       <type>random</type>
     </initialize>

     <path>random</path>

     <interaction>
       <type>general</type>
       <beta>0.5</beta>
       <input-file>dSeq_10_10.dat</input-file>
     </interaction>

     <neighbourhood>  <!-- active only if interaction type is not ‘general’ -->
       <type>box</type>
       <nx>10</nx>
       <ny>10</ny>
       <nz>0</nz>
       <shiftx>0</shiftx>
       <shifty>0</shifty>
       <shiftz>0</shiftz>
       <radius>1</radius>
       <max>3001</max>
     </neighbourhood>

     <output>
       <measurements>measurementlog.txt</measurements>
       <result format="sgems">result.dat</result>
       <temp format="sgems">tmpresult.dat</temp>
       <gridfreq>50</gridfreq>
       <writefreq>10</writefreq>
     </output>
   </action>
</mrf>
```

B Parsing Xml files with NRlib

The MRF package takes user input from an ascii file using the Extensible Markup Language XML.

An xml file has a strict tree structure consisting of nested elements, some of which have attributes and content. An element typically consists of two tags, a start tag and an end tag, possibly surrounding text and other elements. The start tag consists of a name surrounded by angle brackets, like `<step>`; the end tag consists of the same name surrounded by angle brackets, but with a forward slash preceding the name, like `</step>`.

For output to screen and log file the MRF model file syntax is as follows:

```
<messages>
<logfile>
  <level>5</level>
  <name>logs/mrf.log</name>
</logfile>

<screen>
  <level>5</level>
</screen>
</messages>
```

The parsing function takes a reference to its appropriate action element as input. It could be regarded as a pointer to a particular element in the xml file. Below we list some illustrative parts of the **Parse** function in the **FaciesSimulation** class.

The idea is to parse the whole file and then report errors or warnings right before we return. To this end we use a string variable to collect error and warning messages and a status variable to keep track of the keyword status for each keyword as well as the overall parsing status.

```
bool FaciesSimulation::Parse(const Xml::Element& element) {
  Logger& log = Singleton<Logger>::Instance();
  log.Message(Logger::INFO) << "Parsing action \'"
                            << GetName() << "\'" << std::endl;

  /* Declare keyword and overall status variables,
     and initialize the latter
  */
  Xml::ParsingStatus kw_status, status = Xml::PerfectModel;
  std::string errors;
```

For direct decendants of the action element, access of individual input data is done by use of **XmlParser::GetChildArgument** or **XmlParser::RequireChildArgument** depending on whether the element is optional or necessary. Optional elements have a default value. In this example we assume that default values and list of allowed values are defined in a **Definitions** class.

Let us first try to read in the annealing keywords.

```
<annealing>
    <max-temperature></max-temperature>
    <temperature-reduction-factor>0.9</temperature-reduction-factor>
    <steps>10</steps>
</annealing>
```

```
  const Xml::Element* annealing_element =
    element.FirstChildElement("annealing");

  if (annealing_element) {
    kw_status = XmlParser::
      GetChildArgument(*annealing_element,
                       "max-temperature",
                       annealing_temperature_,
                       errors,
                       Default::GetAnnealingTemperature());
    status = std::max(status, kw_status);

    kw_status =  XmlParser::
      GetChildArgument(*annealing_element,
                       "temperature-reduction-factor",
                       annealing_epsilon_,
                       errors,
                       Default::GetAnnealingEpsilon());
    status = std::max(status, kw_status);

    kw_status =  XmlParser::
      GetChildArgument(*annealing_element,
                       "steps",
                       annealing_steps_,
                       errors,
                       Default::GetAnnealingSteps());
    status = std::max(status, kw_status);
  }
  // else using default values from constructor
```

For all of the annealing parameters there are default values defined in `default.h`. These values are used in the constructor, and hence if no <annealing> element is found, default values are used. Likewise, if for example <steps> is missing, its default is used.

Sometimes we need to read a group of data with variable number of input elements.

```
<lithology>
  <facies>
      <name>sand</name>
      <fraction>0.55</fraction>
  </facies>
  <facies>
      <fraction>0.45</fraction>
      <name>shale</name>
  </facies>
</lithology>
```

Note that the order of subelements for “name” and “fraction” is arbitrary.

```
  // For nested child elements
  {
    const Xml::Element* lithology_element =
      element.FirstChildElement("lithology");

    if (!lithology_element) {
      errors += std::string("ERROR: ") +
        "\'lithology\' not found in action \'" +
        GetName() + "\'.\n";
      status = Xml::ModelFailure;
    }
    else {
      // Get all children
      const Xml::Element* facies_element =
        lithology_element -> FirstChildElement("facies");
      for (; facies_element;
             facies_element =
                 facies_element -> NextSiblingElement("facies")) {
        std::string name("");
        double fraction(0);
        kw_status = XmlParser::GetChildArgument(*facies_element,
                                                "name",
                                                name,
                                                errors);

        facies_.push_back(name);
        status = std::max(status, kw_status);

        kw_status =
          XmlParser::GetChildArgument(*facies_element,
                                      "fraction", fraction,
                                      errors);
        target_facies_volume_fraction_.push_back(fraction);
        status = std::max(status, kw_status);
      }
    }
  }
```

Sometimes a program parameter has a predefined list of allowed values. In the program we may want to represent these with enums. In the input file, for the sake of user-friendlieness, we must use strings. Hence, we have a list of allowed input string values.

```
{
    // Read method
    std::string text;
    kw_status = XmlParser::
        GetChildArgument(element,
                         "method",
                         text, // read_string
                         errors,
                         Default::GetMethodName(), // default
                         true, // Lowercase flag
```

```
                          &Definitions::GetSimulationModes()
                          );
    status = std::max(status, kw_status);
    /*
      Now we have an allowed string.
      Loop over the allowed values,
      and find the corresponding enum value
      represented by an unsigned integer.
    */
    std::vector<std::string>::const_iterator v;
    unsigned int mode = 0;
    for (v = Definitions::GetSimulationModes().begin();
         v != Definitions::GetSimulationModes().end(); ++v, ++mode) {
      if (*v == text) {
        break;
      }
    }
    // cast to enum
    method_ = static_cast<Definitions::SimulationMode>(mode);
}
```

Consider the “result” keyword below. Assume that it is required by the program.

```
<output>
    <measurements>measurementlog.txt</measurements>
    <result format="sgems">result.dat</result>
    <temp format="sgems">tmpresult.dat</temp>
    <gridfreq>500000</gridfreq>
    <writefreq>10000</writefreq>
</output>
```

In order to read it, we first have to locate the “output” element.

```
const Xml::Element* output_element =
  element.FirstChildElement("output");
if (!output_element) {
  errors += std::string("ERROR: ") +
    "\'output\' not found in action \'" +
    GetName() + "\'.\n";
  status = Xml::ModelFailure;
}
else {     // Output element is present
  kw_status =
        XmlParser::RequireChildArgument(*output_element,
                                        "result",
                                        facies_result_file_,
                                        errors);
  status = std::max(kw_status, status);

  std::string text;
  kw_status = XmlParser::
        RequireChildAttribute(*output_element,
                              "result",
                              "format",
                              text,
                              errors,
                              true, // lowercase
                             &Definitions::GetFileFormatNames());
  std::vector<std::string>::const_iterator v;
  unsigned int mode = 0;
  for (v  = Definitions::GetFileFormatNames().begin();
       v != Definitions::GetFileFormatNames().end(); ++v, ++mode) {
    if (*v == text) {
      break;
    }
  }
  facies_result_format_ = static_cast<Definitions::FaciesFormat>(mode);
  status = std::max(kw_status, status);
}
```

At the end of the parsing function, we check the status and print out error or warning messages as required. Depending on the status we return true for success and false for failure.

```
  // Check final parsing status and write out any errors.
  if (status > Xml::ModelWillDo) {
    log.Message(Logger::SEVERE) << errors;
    return false; // failure
  }
  else if (status > Xml::PerfectModel) {
    log.Message(Logger::WARNING) << errors;
  }
  return true; // success
}
```