The Frequency Matching Method: A Fortran Implementation

Katrine Lange[a,b,*], Jan Frydendall[a,b]

[a] *DTU Informatics, Technical University of Denmark, Asmussens Allé, 2800 Kgs. Lyngby, Denmark*
[b] *Center of Energy Resources Engineering, Technical University of Denmark, Søltofts Plads, 2800 Kgs. Lyngby, Denmark*

Abstract

In the last decade multiple point statistics has become increasingly popular as a tool for incorporating complex prior information when solving severely under-determined inverse problems in geosciences. A variety of methods doing such have been proposed but often the implementation of these is not straightforward. This paper discusses the aspects of the implementation of the recently proposed Frequency Matching Method and how it is made computationally feasible also for larger problem instances. The source code has been made publicly available WHERE??, and this paper also provides an example of how to apply the Frequency Matching method to a linear inverse problem.

Keywords: multiple points statistics, training image, maximum a posteriori model

1. The Frequency Matching Method

The Frequency Matching (FM) method is a method for computing the maximum a posteriori model of an inverse problem using multiple point statistics learned from a training image (TI) as prior information. Inverse problems having such type of a priori information arises in many fields where they are concerned with modelling unknown parameters describing spatial properties. They are typical in geosciences where for instance a property of

*Corresponding author
Email address: katla@imm.dtu.dk (Katrine Lange)

the subsurface of the Earth should be modelled. The available data is often scarce resulting in the inverse problems being severely under-determined. This is handled but taking into consideration prior information about the structures of the subsurface, and when modelling spatial properties this can be available in so-called training images.

Models of parameters describing spatial properties are often referred to as images, as one can imagine the one-dimensional vector of model parameters reshaped to a two or three-dimensional image, letting the colours of the image illustrate the values of the property they are describing.

The method was first published in Lange et al. (2012). This manual describes a Fortran implementation of the FM method for a linear inverse problem. The manual is part of the thesis Lange (2013), and we refer hereto for a much more detailed motivation for the method and discussion of the use of multiple point statistics when solving inverse problems in the geosciences. The purpose of the manual is to make other users capable of understanding and using the Fortran version of the FM method on their respective problems. The derivation of the FM method itself is therefore left out but can be found in the reference list.

The FM method itself has no limitations when it comes to non-linearity but the current implementation assumes that the inverse problem is linear, i.e., its associated forward problem is linear:

$$\mathbf{G}\mathbf{m} \quad = \quad \mathbf{d}^{\text{obs}}, \tag{1}$$

where $\mathbf{G}$ is a known system matrix, $\mathbf{d}^{\text{obs}}$ is a set of observed data values and $\mathbf{m}$ is the model parameters to be determined. The FM assumes that these model parameters can take on a limited number of categorical values. Often the model parameters are binary, for instance, when modelling the flow of the subsurface the model parameters can either be 0 which represent an zones with high permeability and therefore easy flow, or 1, which represents low-permeable zones. Let $\mathsf{sV} + 1$ be the number of categories voxel values can belong to, i.e., for a binary image $\mathsf{sV} = 1$.

The FM method computes the maximum a posteriori model, i.e., the model with maximum a posteriori probability, using multiple points statistics learned from a training image as a priori information. To do so it formulates a closed form expression of the a priori probability density function of models, which can be interpreted as a distance measure between a model/image and a training image. The distance expresses how dissimilar the multiple point

statistics of the images are. Models with multiple point statistics similar to the multiple point statistics of the training image have short distances to the training image and they are therefore assigned high probability, and likewise models with dissimilar multiple point statistics will have large distances to the training image and they will therefore have low probability. Using Bayesian inverse problem theory the posterior probability density function is proportional to the product of the prior and the likelihood function. For more details on probabilistic inversion theory see for instance Tarantola (2005).

The current implementation of the FM method assumes that the inverse problem is linear. The FM model is then computed as the minimizer of the logarithm of the posterior:

$$\mathbf{m}^{\mathrm{FM}} = \underset{\mathbf{m}}{\operatorname{argmin}} \left\{ \frac{1}{2} \|\mathbf{d}^{\mathrm{obs}} - \mathbf{G}\mathbf{m}\|^2_{\mathbf{C}_\mathbf{d}^{-1}} + \alpha^2 c(\mathbf{m}) \right\}, \tag{2}$$

where c is the distance function to the training image based on dissimilarities in multiple point statistics, and α is a positive weighting parameter. The first term of this objective function to be minimized is then a data misfit term coming from the likelihood function of the model parameters given the observed data. The second term comes from the prior; it is the dissimilarity distance previously introduced.

The multiple point statistics of an image is represented by what the FM method defines as the frequency distribution. This is basically just a histogram of the counts of the different patterns found in the image, where the patterns, if their size is chosen wisely, are assumed to describe the multiple point statistics of the image. For this reason, we might occasionally refer to frequency distributions as histograms.

The current Fortran implementations makes use of the assumption of linearity in the forward problem from Eq.(1). As mentioned, the FM method also works for non-linear problems, and we will in our presentation of the procedures point out where changes can be made such that the implementation can be used with any non-linear problem.

2. Aspects of the Implementation

The FM model is computed as the minimizer of the logarithm to the posterior probability, given by (2). The minimization is performed using simulated annealing. We will not go into details about the choice of this optimization method or the method itself, but for more information on simulated annealing see Kirkpatrick et al. (1983). Pseudo code for applying simulated

annealing to the FM method can be seen in Lange et al. (2012), and Lange (2013) contains a discussion of many of the details of the derivation of the FM method.

Figure 1 shows an overview of the most important interactions between the Fortran procedures in the implementation of the FM method. For now we will provide a short walk-through of the procedures, and a short description of what each of them do is provided in section Appendix A.

The implementation is based on the FMM procedure, which primary function is to set up and initialize all the inputs for the FM method and the simulated annealing scheme. The FMM procedure also has the task of extracting multiple point statistics of the training image representing it using the—for that purpose designed—tree structure and generate its frequency distribution. This is the only procedure that uses the training image itself, and it only passes on its tree and frequency distribution.

The simulated annealing is implemented in the CompOptimalImage procedure. Provided with an initial model the first thing the CompOptimalImage procedure does, is to extract the multiple point statistics of the initial model (InferTree) and compute its frequency distribution (Tree2Hist). It then computes the objective function value of the initial model (CompObjFun). The CompObjFun procedure calls two other procedures, CompChiDist and CompDataFit, to compute each of the two terms of the objective function.

Vertical arrows in the figure represent loops, and in the CompOptimalImage procedure these are used to loop through the iterations of the simulated annealing algorithm.

For each iteration a perturbed image is generated (SimNewImage) which is done by erasing the voxel values in a part of the image and then re-simulating them using sequential simulation with the multiple point statistics learned from the training image (SimVoxel). Each time a voxel has been re-simulated the tree should be updated to fit the new image (UpdateTree). Here again the vertical arrows represents loops indicating that these two tasks are done voxel by voxel.

Afterwards the frequency distribution of the perturbed image is computed (again Tree2Hist). The objective function value is then computed of the perturbed image (again CompObjFun) and finally, the perturbed model is, maybe, accepted and the variables updated accordingly (UpdateSA).

The procedures that have been left out of the diagram are auxiliary procedures that are mostly related to operations on trees.

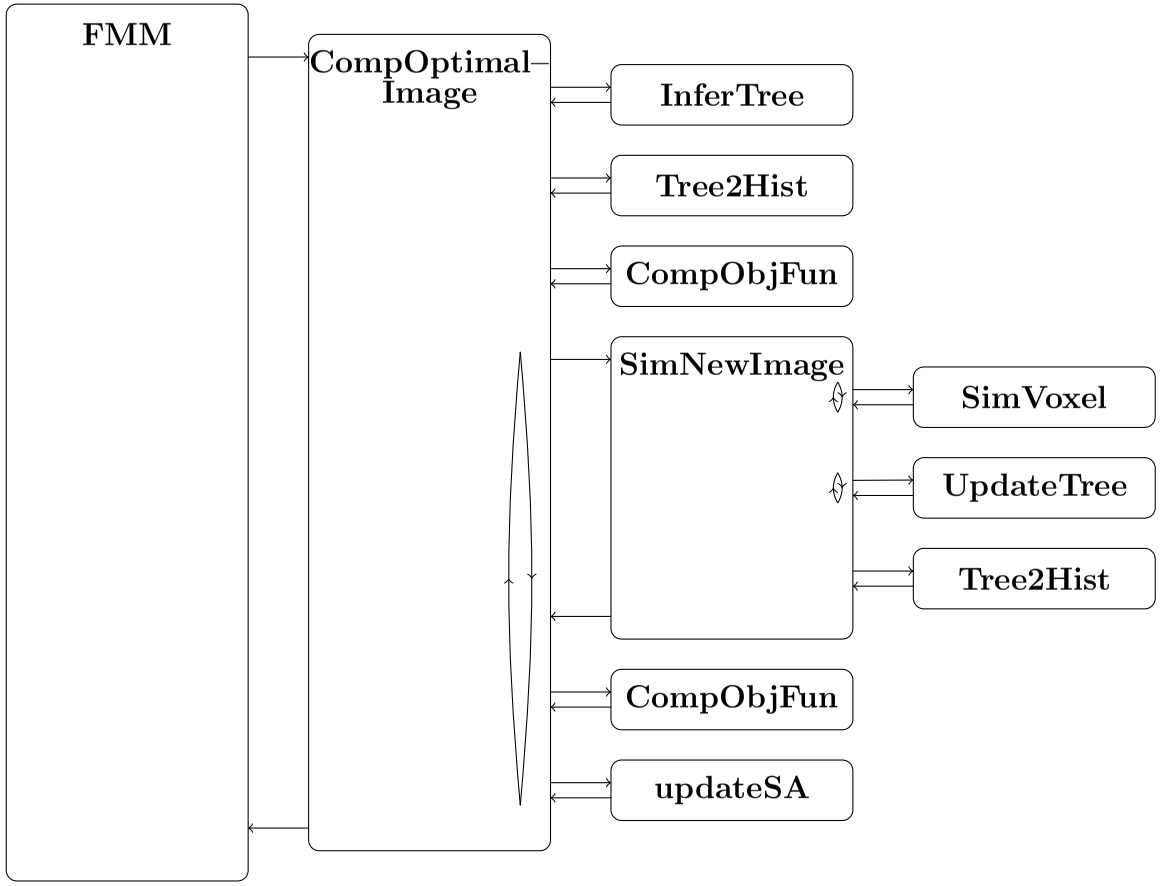


Figure 1: Overall structure of the FM implementation.

2.1. Fortran Version

TODO: add which version was used, which compilers, and where it has been tested.

2.2. The Inverse Problem

The FMM procedure takes among other inputs the parameters from Eq.(1). For that we have defined a structure callled inverseproblem. It contains the following parameters:

G: 2D array holding the system matrix $\mathbf{G}$.

dobs: 1D array holding the vector of data observations $\mathbf{d}^{\text{obs}}$.

invCov: 2D array with the inverse of the covariance matrix, $\mathbf{C}_{\mathbf{d}}^{-1}$.

cat: 1D array with sV+1 elements, one for each category of voxel values. This is used to transform the images with categorical voxel values to models with physical parameter values. The model parameter associated with a voxel with value i will be assigned the value cat(i + 1) in order to compute the data fit.

The parameters specifying the inverse problem is being passed along between procedures in the variable InvProb. These can be used to specify any linear, inverse problem, and the implementation can therefore also be used to solve inverse problems with applications outside the field of geosciences.

2.3. Specifying Neighbourhoods

A pattern is defined by the value of a voxel and the values of its neighbouring voxels. This means, we have to distinguish between two types of voxels in an image: inner voxels and non-inner voxels. Inner voxels are those that are sufficiently far away from the boundaries that the have enough neighbouring voxels to make a pattern. Non-inner voxels are the rest, i.e., those that are close to the boundary and therefore do not have as many neighbouring voxels around them. How far away from the boundary a voxel has to be in order to be an inner voxel depends on how big the neighbourhood of voxels are defined to be and thereby how big the patterns become.

Several parameters are needed to specify the dimensions of neighbourhoods and these are collected in a special type of structure called NeighborMask. It contains the following parameters:

mat: 3D integer array of the shape of a neighbourhood of an inner voxel. Voxels in positions where the element of mat is 1 is included in the

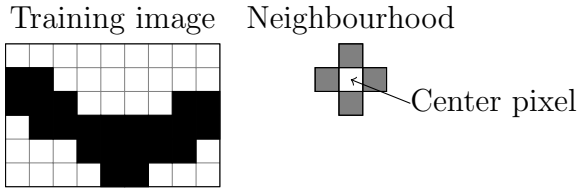


Figure 2: Tiny training image used for illustration purposes. A neighbourhood is chosen to be the upto four nearest neighbours as shows to the right. The pixels within the neighbourhood is numbered from top to bottom from left to right, how to define this neighbourhood is explained in the text. The tree of this training image can be seen in Figure 4.

.

neighbourhood, and voxels where the corresponding element is 0 is not included. The value corresponding to the center voxel does not matter.

nc, mc, pc: integers denoting the coordinate in the mat array of the center voxel of a neighbourhood.

n, m, c: Dimensions of the patterns, i.e., the mat array is n × m × p.

nodes: 2D integer array with relative coordinates from the center voxel to each of its neighbours. That means, given the image coordinates of a voxel, nodes can be used to compute the image coordinates of all of the neighbours of that voxel. The array has a row for each neighbour and three columns that holds the coordinates in each dimension.

Neighbourhoods can be defined in two ways. If the patterns have the simple shape of a hyper-rectangle, the user can simply specify the dimensions of the patterns, m, n and p. These must be odd numbers and the center of the patterns is then assumed to be directly in the middle. This is probably the most common type of pattern to use. If the patters are not that simple, the user can directly specify the mat array and its center coordinate, (nc, mc, pc). The dimensions of the mat array do not have to be odd, and the center can be located anywhere.

The parameter sN is used throughout the procedures as the number of voxels in a pattern excluding the center voxel. This means that nodes has sN rows. In the procedures the structure specifying the neighbourhoods is denoted Nmask.

Figure 2 shows a tiny example of a training image and a neighbourhood that could be used to describe the multiple point statistics of it. To define this non-rectangular neighbourhood the user will need to specify the mat array:

$$\mathsf{Nmask\%mat} \;=\; \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

and its center coordinates:

$$\begin{aligned} \mathsf{Nmask\%nc} &= 2, \\ \mathsf{Nmask\%mc} &= 2, \\ \mathsf{Nmask\%pc} &= 1. \end{aligned}$$

Then the FMM will derive the dimensions of the patterns:

$$\begin{aligned} \mathsf{Nmask\%n} &= 3, \\ \mathsf{Nmask\%m} &= 3, \\ \mathsf{Nmask\%p} &= 1, \end{aligned}$$

leading to the following relative row, column and layer coordinates of neighbours:

$$\mathsf{Nmask\%nodes} \;=\; \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Keep in mind, that even for 2D images the third dimension does exist, it will only have the size one.

2.4. The Tree Structure

A tree is a complex structure and most vital for the computational feasibility of the implementation. It is therefore one of the features we will elaborate on. The purpose of the tree is to extract the patterns of a training image, and then use them to describe the multiple point statistics of the training image.

The tree is a structure of the type linked list. This is the same approach that was applied in the SNESIM algorithm (Strebelle, 2002) when creating

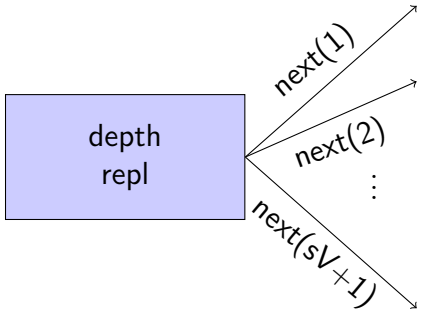


Figure 3: Illustration of a node of a tree as represented by the streenode structure.

search trees in order to overcome the problems of generating and storing large data bases of patterns. In our implementation trees are furthermore used to easily generate frequency distributions, which is the actual input to the dissimilarity function c from Eq.(2).

We like to think of trees as consisting of a set of nodes and edges as a tree in the mathematical sense of a graph. A tree, T, is based on a root node, and it is often that node we pass along the different procedures. From this root node we can navigate deeper into the tree via links, which in this case are pointers. But keep in mind that the T parameter just contains a node. The node is often the root node though, and in those cases we will refer to it as a tree.

A node of a tree is defined as a structure streenode containing three variables:

depth: integer defining how deep into the tree this node is located.

repl: real array with sV+1 elements. The array holds the count of patterns in the image that has a certain structure. This structure is dependent on the depth of the root, and the deeper into the tree a node is located, the more voxels of the patterns is used to define the structure.

next: array of pointers with sV+1 elements, these are the links to nodes placed a level deeper into the tree.

Figure 3 shows the structure defined to describe a node. A tree of an image is constructed by first creating the root node and then adding new nodes as the image is being scanned and new patterns found. Assigning the root node

depth level 0, the maximum depth of a tree is sN, and at any depth i there can maximum be $(\mathsf{sV}+1)^i$ nodes.

Let T denote the root node of a tree; this node contains information of the center values themselves, not including any neighbouring voxel values. We define the T%repl array to hold the unscaled distribution of voxel values of inner voxels in the image, so simply the counts of how many inner voxels in the image have each of the values $0, 1, \ldots, \mathsf{sV}$. Notice how T%repl(k) holds the count of voxels with value $k-1$. The pointers in the array T%next points to sV+1 new nodes of the tree. These nodes are at depth level 1 and they therefore contain information about center voxels taking into consideration the value of their first neighbouring voxel. As the first neighbouring voxel can have sV+1 different values we need the same number of pointers to cover all cases. This means, the pointer T%next(i) points to the node representing all partial patterns, where the first neighbouring voxel has value i−1. This unscaled distribution for all values of center voxels will be stored in the repl array of that node. It can be accessed by T%next(i)%repl. This means the element T%next(i)%repl(k) holds the counts of patterns in the image where the first neighbouring voxel has the value $i-1$ and the center voxel has value $k-1$. And in the same manner, the jth pointer of this node, T%next(i)%next(j), points to the node patterns where the values of the two first neighbouring voxels are $i-1$ and $j-1$, respectively. This means, the node holds the unscaled conditional probability distribution of the value of a center voxel given these specific values of the first two neighbouring voxels. This is repeated until depth level sN, where all sN neighbouring voxels have been included in the partial structure, that each node represents.

To sum up, an arbitrary node T provides us the following information:

T%depth: depth level in the tree where the node is placed.

T%repl(k): count of a specific partial pattern in the image with center value $k-1$. The partial pattern is unknown to this node, but the values of the first T%depth neighbouring voxels are given by the location of the node in the tree.

T%next(i) pointer to the node with the same partial pattern as the current but where the T%depth+1 voxel has value i.

Notice how a node T cannot give us any information about the partial patterns it represents. From a node we can only extract information that are

deeper into the tree and not information that belong on a previous level. The pointers in T%next or one-way streets, so to speak, and the tree contains no pointers sending us in the opposite direction. This is not needed in the implementation.

The frequency distribution of an image is the (unscaled) distribution of patterns. As the bottom level of the tree contains exactly the counts of patterns with all of the possible combination of center and neighbourhood voxel values, the frequency distribution can simply be constructed by combining all level sN repl arrays.

Recall, Figure 2 shows an example of a training image and an example of a neighbourhood. This is a very tiny training image and neighbourhood chosen for illustration purposes only. The resulting tree can be seen in Figure 4. The frequency distribution of the image is constructed by extracting all 4th level repl arrays of the tree.

On each node is written the values of its **repl** array, and on each edge is written the colour of the neighbouring voxel represented by the current depth level. Black voxels are assigned the value 0 and white have the value 1.

To compare an image to a training image a tree describing its multiple point statistics must be derived and its frequency distribution determined. This allows for the evaluation of c. However, constructing the tree is done in a different manor than for a training image itself. when constructing the tree we take advantage of the fact that the dissimilarity function c depends only on the patterns of the image that also appear in the training image. We therefore generate not the tree containing all patterns found in the image but only those patterns that are also found in the training image. That means, the tree of the image will have the same shape (same nodes and edges) as the tree of the training image. This makes the frequency distribution consisting of all bottom level repl arrays directly comparable to the frequency distribution of the training image, as they, element by element, describe the count of identical patterns in the two images.

2.5. Perturbation Domain

A set of parameters is needed to specify the size of the domain of an image that needs to be erased and re-simulated to perturb the image. The domain is assumed to be hyperrectangular. To define it we have the DomainMask structure that contains the following parameters:

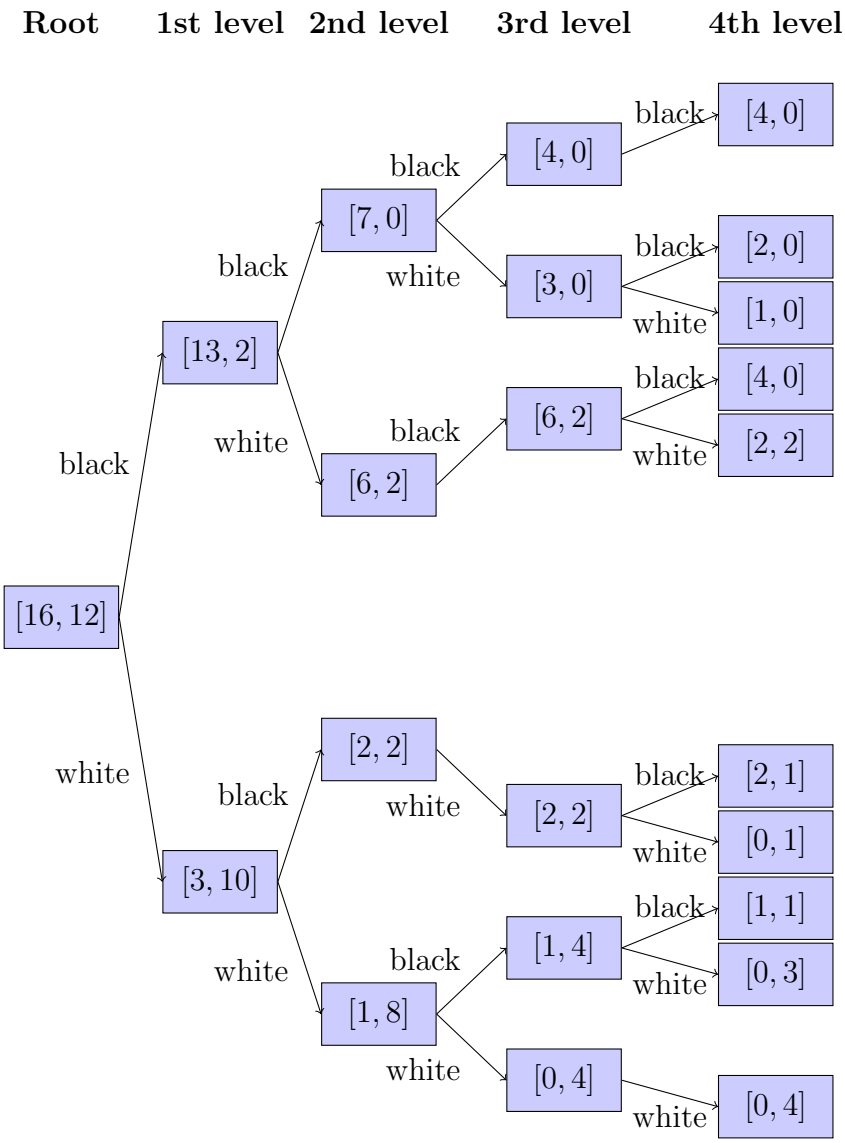


Figure 4: The tree structure of the tiny training image in Figure 2 using a neighbourhood consisting of the four closest voxels as shown in the figure. We assign black voxels the value of 0 and white voxels the value of 1. Notice how the count of each pattern is represented in the three, for instance, the number of black pixels with all black neighbouring pixels is 4, which is found by starting in the root node, following all edges labelled black until reaching the bottom level, and then accessing the first element of the repl array. Also notice how for every single node, if you sum the repl arrays of the nodes it is pointing to, you get the repl array of the node itself. This means all repl arrays only hold counts of inner pixels.

n, m, p: integers, dimensions of the domain to be re-simulated. These must be odd numbers.

nc, mc, pc: integers, coordinates of the center of the domain.

nodes: 2D integer array with relative coordinates from the center voxel of the domain to each other voxel in the domain.

mat: 1D integer array with sN elements. This array holds a distance from the center voxel of a neighbourhood to each of its neighbouring voxels. It is used in the re-simulation to determine on which voxels the simulation should be conditioned.

The domain structure is referred to as Dblock in the procedures. Like for the neighbourhood mask the user do not need to specify most of its parameters. In fact one should only decide on the dimensions of the block Dblock&n, Dblock&m, Dblock&p, and the FMM procedure then constructs the remaining. For the distance array is used the L_1-norm.

2.6. Optimization Options

Simulated annealing is used as the solution method to the optimization problem defining the $\mathbf{m}^{\mathrm{FM}}$, and to hold the parameters used by the algorithm we have the type option. It holds the following parameters:
t0: real number, the initial temperature.

tmin: real number, the final temperature.

maxIter: integer, maximum number of iterations allowed to be used per voxel parameter.

runs: integer, number of times to run the simulated annealing algorithm.

multigrid: integer, number of multigrids to use.

condopt: logical, in case of multiple grids used, it determines if the solution on a fine grid should be conditioned on the optimal solution from the coarser grid (condopt = true) or not (condopt = false).

The simulated annealing uses an exponential cooling rate that is calculated such that the number of iterations allowed is exactly the number of iterations used. The implementation of the FM method has been prepared for multiple

grid simulation but these are not yet implemented. This can be done by implementing a loop in the FMM procedure such that CompOptimalImage will be called with different grids. Also in some cases it can be beneficiary to restart the simulated annealing algorithm and although this option has not yet been implemented the code has been prepared. Here a loop can be inserted in the CompOptimalImage procedure so that the simulated annealing scheme is run multiple times.

3. Example: Crosshole Travel Time Tomography

As an example of use of the frequency matching method we will show how to solve a synthetic crosshole travel time tomography problem similar to the one described in Lange et al. (2012). Crosshole travel time tomography involves the measurement of seismic travel times between two or more boreholes in order to determine an image of seismic velocities in the intervening subsurface. Seismic energy is released from sources located in one borehole and recorded at multiple receiver locations in another borehole. In this way a dense tomographic data set that covers the interborehole region is obtained. We will create a synthetic test case based on a setup with two vertical boreholes. The horizontal distance between them is 500 meters and they each of the depth of 500 meters. The two-dimensional vertical domain between the boreholes is divided into 120 times 50 quadratic cells. The seismic velocity is assumed constant within each cell. The model parameters of the problem are these propagation speeds, meaning the problem has 6000 unknown model parameters. The observed data is the recorded first arrival times from the seismic signals. In each borehole are placed 12 equally distributed sources and 48 equally distributed receivers. We assume a linear relation between the data observations and the model parameters. The sensitivity of a seismic signals is simulated as straight rays.

It is assumed that the interborehole region consists of a background with slow propagation speed and a horizontal channel structure of zones of high propagation speed. The speeds are chosen as 1600 meters per second and 2000 meters per second, respectively. The a priori knowledge of the channel structure is assumed described by the training image in Figure 5. We have chosen to let the neighbourhood of a pixel consist of its 36 closest neighbours.

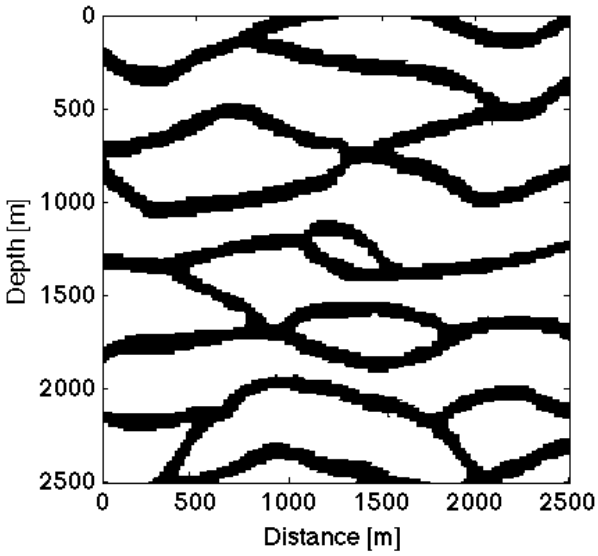
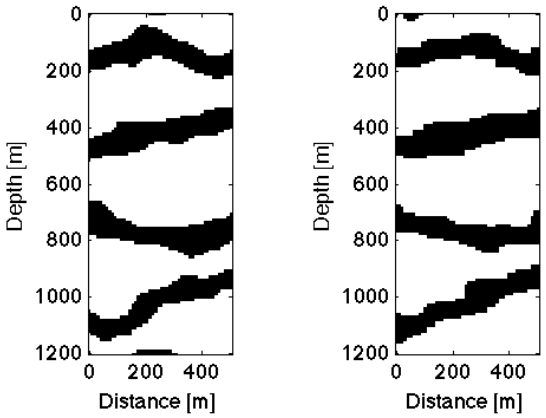


Figure 5: Training image (resolution: 250 by 250 pixels).



(a) Reference model. (b) Optimal model.

Figure 6: Reference model for the synthetic crosshole travel time tomography example and its computed optimal solution. The resolution is 120 by 50 pixels. The solution computed by the FM method is the model with maximum a posteriori probability.

This is specified by:

$$\mathsf{Nmask\%mat} \;=\; \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}.$$

A reference model is generated based on the training image using the SNESIM (Strebelle, 2002) algorithm. The first arrival times of the reference model is simulated. These are then added 5% relative Gaussian noise and assumed to be the observed data. Figure 6a shows the reference model.

The forward problem: Voxels belonging to the background are assigned to the category 0 and voxels belonging to the zones of high propagation speed are assigned the category 1. The forward problem is linear in the inverse of the propagation speeds, i.e., the physical values of the voxel values of the two categories are specified as $\mathsf{cat} = [1/1600, 1/2000]$.

The observed data and the coefficent matrix from the forward problem is generated using MATLAB. They have then been stored in a text file that can be read into Fortran using the standard `read` routine and then save in the parameters `dobs` and `Gmat`, respectively. The problem has 1152 data observations and 6000 model parameters so it is severely under-determined.

The noise added to the reference model is independent and has estimated standard deviation $\hat{\sigma} = 1.9141 \cdot 10^{-2}$. This yields the data covariance matrix $\mathbf{C_d} = \hat{\sigma}^2 I$, where I is the identity matrix. The inverse data covariance matrix `invCov` is then defined as the inverse of this.

The prior term: The weighting constant multiplied to the prior term is chosen as $\alpha = 10^{-1}$, which means $\mathsf{alpha} = 10^{-2}$.

Perturbation of images: The domain of voxels to be resimulated when creating a perturbed image is chosen as $\mathsf{nD} = 13$, $\mathsf{mD} = 13$ and $\mathsf{pD} = 1$.

Optimisation parameters: The cooling rate is defined by the starting temperature $\mathsf{t0} = 10^2$ and the minimum temperature $\mathsf{tmin} = 10^{-5}$. The

simulated annealing algorithm is allowed to use iter = 0.5 for each of the 6000 pixels in the image.

Computing the optimal model using the allowed 3000 iterations took approximately 16 minutes on a Macbook Pro 2.66 GHz Intel Core 2 Duo with 4 GB RAM.

The computed optimal model is shown next to the reference model in Figure 6. It is seen how it correctly locates the channels. The width and curvature of the channels also clearly resembles those of the reference model. We therefore conclude that the choice of weighting constant alpha and the optimisation parameters were suitable for the problem at hand, and the example successfully illustrate how the Fortran implementation of the frequency matching method can be applied.

List of Appendices

Appendix A. Primary Procedures

The following is a list of the primary Fortran procedures in the FM implementation. For each procedure its purpose is briefly explained, and if there are any particularly interesting details in the implementation they are discussed. Also a list of input and/or output variables is provided. The lists hold the variable name, a short description of its use and its type.

To see which procedures call others we refer to Figure 1. The procedures are listed in the order they are called which also appears from the figure. Auxiliary procedures are listed in section Appendix B.

Appendix A.1. FMM

This procedure act as an intermediary between the user specified input parameters and the implemented FM method. It sets up all the necessary inputs for the simulated annealing scheme based on the user inputs and reads the multiple points statistics from the training image. The procedure then calls the CompOptimalImage procedure to compute the FM model (2). This model along with its frequency distribution, the frequency distribution of the training image and other parameters of special interest are written to files. These can later to loaded into for instance MATLAB to visualize the results.

In case the code is modified to handle multiple grids this would be a suitable procedure in which to loop over the grid levels, and for each level set up the corresponding tree of the training image and do the conversion from coarse to fine grid before calling the CompOptimalImage.

Variable	Description	Type
Z0	Initial image, i.e., the starting image for the iterative solution method. This image should satisfy hard data constraints, if any.	3D integer array
Ztrain	Training image.	3D integer array
Zcond	This array has the same dimensions as the image Z0, and it is used to state if any of the voxels should satisfy hard data constraints. If there are no hard data constraints the array should be all false. If some voxels are only allowed to take on a specific value the corresponding element of Zcond should be true.	3D logical array

Nmask	Contains the parameters that define the neighbourhoods.	structure
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
InvProb	Parameters specifying the linear inverse problem.	structure
alpha	Weighting parameter α^2 of the prior term in the objective function.	real
Dblock	Contains the parameters that define the domain of an image to be re-simulated to create a perturbed image.	structure
options	Parameters for the simulated annealing scheme.	structure
solutions	Structure holding the results from the simulated annealing. These are written to files.	structure
Ps	Holds the values of each of the two terms in the objective function for each iteration of the simulated annealing algorithm.	2D real array

Appendix A.2. *InferTrainTree*

This procedure generates the tree, Ttrain, describing the multiple point statistics of a training image, Ztrain. Patterns are extracted one by one from the training image and added to the tree. The frequency distribution of the training image is later extracted from the tree by the procedure Tree2Hist.

Variable	**Description**	**Type**
Ztrain	Training image.	3D integer array
nodes	Array with sN rows and three columns. Each row contains the relative coordinates (in the three columns respectively) from a center voxel to one of its neighbouring voxels.	2D integer array
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer

sN	Parameter, number of neighbours for an inner voxel; which means patterns have sN + 1 voxels.	integer
Ttrain	Tree of patterns extracted from the training image.	tree

Appendix A.3. *Tree2Hist*

This procedure constructs the frequency distribution (or the histogram), H, of an image given its tree, T. The frequency distribution is a two dimensional array with sV + 1 rows and a column for each combination of voxel values in a neighbourhood. The ith row has the count of the appearances of the different neighbourhoods with center voxel having the value $i-1$.

This format has the advantage that each column of the frequency distribution is the unscaled conditional probability distribution of the value of center voxel given the values of its neighbouring voxels. These conditional distributions used for instance for re-simulating voxel values are therefore easily accessible. The disadvantage is that we might store more zero elements than necessary although no more than sV times too many.

Variable	**Description**	**Type**
sV	Parameter, the images contain sV + 1 categories of voxel values, namely $0, 1, \ldots,$ sV.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have sN + 1 voxels.	integer
T	Tree of patterns extracted from an image Z	tree
H	Frequency distribution of an image Z.	2D real array

Appendix A.4. *CompOptimalImage*

This procedure is the most central in the implementation of the FM method. It is the one that solves the inverse problem by use of a simulated annealing algorithm and it therefore takes several inputs. Everything from the starting image and of course the multiple point statistics learned from

a training image to the FM parameters specifying the neighbourhoods and the parameters associated with simulating perturbed images. It returns the computed optimal image, $\mathbf{m}^{\mathrm{FM}}$, as well as the frequency distribution of it. To check the convergence of the simulated annealing algorithm it also returns the objective function values for all iterations.

Variable	**Description**	**Type**
Z0	Initial image, i.e., the starting image for the iterative solution method. This image should satisfy hard data constraints, if any.	3D integer array
Zcond	This array has the same dimensions as the image Z0, and it is used to state if any of the voxels should satisfy hard data constraints. If there are no hard data constraints the array should be all false. If some voxels are only allowed to take on a specific value the corresponding element of Zcond should be true.	3D logical array
Ttrain	Tree of patterns extracted from the training image.	tree
Htrain	Frequency distribution of the training image Ztrain.	2D real array
nodes	Array with sN rows and three columns. Each row contains the relative coordinates (in the three columns respectively) from a center voxel to one of its neighbouring voxels.	2D integer array
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
InvProb	Parameters specifying the linear inverse problem.	structure
alpha	Weighting parameter α^2 of the prior term in the objective function.	real
Dblock	Contains the parameters that define the domain of an image to be re-simulated to create a perturbed image.	structure

options	Parameters for the simulated annealing scheme.	structure
Zopt	The optimal image, the $\mathbf{m}^{\mathrm{FM}}$, computed by the simulated annealing algorithm.	3D integer array
Hopt	Frequency distribution of the optimal image Zopt.	2D real array
Ps	Holds the values of each of the two terms in the objective function for each iteration of the simulated annealing algorithm.	2D real array

Appendix A.5. *InferTree*

This procedure infers the tree of an image so that its multiple point statistics can be compared to those of a training image. Notice that it therefore takes as input the tree of the training image that the image should later be compared to. This is necessary as the tree should only contain patters also found in the training image. Once the image has been scanned and all patterns that should be stored has been added to the tree, the procedure shapeTree is called, to make sure the tree just generated has the same shape as the tree of the training image. This is needed in order to easily compare their frequency distributions.

Variable	**Description**	**Type**
Z	Image.	3D integer array
nodes	Array with sN rows and three columns. Each row contains the relative coordinates (in the three columns respectively) from a center voxel to one of its neighbouring voxels.	2D integer array
Ttrain	Tree of patterns extracted from the training image.	tree
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
T	Tree of patterns extracted from an image Z	tree

Zex	This array is the same size as the image Z and it is used to indicate which voxels of the image are center of patterns found also in the training image. It is used by the procedure getNewIt to propose where an image should be perturbed.	3D logical array

Appendix A.6. CompObjFun

Given an image this procedure computes the values of each of the terms in the objective function from Eq.(2). To be able to track convergence of the simulated annealing algorithm the terms are not added but instead the procedure returns a two-element array P such that:

$$\begin{aligned} \mathsf{P}(1) &= \frac{1}{2}\|\mathbf{d}^{\text{obs}} - \mathbf{G}\mathbf{m}\|^2_{\mathbf{C}_\mathbf{d}^{-1}}, \\ \mathsf{P}(2) &= \alpha^2 c(\mathbf{m}). \end{aligned}$$

The dissimilarity function c is evaluated by the ompChiDist procedure, and the data misfit is computed by a call to the CompDataFit procedure.

In case the inverse problem is not linear the CompDataFit procedure needs to be replaced.

Variable	**Description**	**Type**
H	Frequency distribution of an image Z.	2D real array
Htrain	Frequency distribution of the training image Ztrain.	2D real array
N	Number of voxels in the image Z.	integer
Z	Image.	3D integer array
InvProb	Parameters specifying the linear inverse problem.	structure
alpha	Weighting parameter α^2 of the prior term in the objective function.	real
P	Two-element array that holds the values of each of the terms in the objective function.	1D real array

Appendix A.7. CompChiDist

This procedure is called by CompObjFun and computes the dissimilarity of an image compared to a training image by computing the distance between their frequency distributions.

Variable	Description	Type
H	Frequency distribution of an image Z.	2D real array
Htrain	Frequency distribution of the training image Ztrain.	2D real array
N	Number of voxels in the image Z.	integer
X	Value of the dissimilarity function of the two frequency distributions.	real

Appendix A.8. CompDataFit

This procedure is called by CompObjFun and computes the data misfit of a model.

Variable	Description	Type
Z	Image.	3D integer array
InvProb	Parameters specifying the linear inverse problem.	structure
L	Value of the data misfit of for the image	real

Appendix A.9. SimNewImage

This procedure is used to perturb images. Provided with the current image from the simulated annealing iteration, Ztest, it will return a new, perturbed image Znew. Znew is generated by erasing the values of a sub-

set of the voxels and then re-simulating them using sequential simulation conditioned on the voxel values of the remaining part of the image.

This procedure takes as input the row, column and layer index of a voxel. The voxel is the center of a domain where voxel values are erased. The values of the voxels in the domain are then re-simulated one by one conditioned on the values of all voxels outside the domain and the already re-simulated values inside the domain. Of course voxels that should satisfy hard data constraints are not allowed to be changed and these are therefore not erased and re-simulated. Their values are kept and instead used to condition the re-simulation on.

The re-simulated values voxels are stored separately. And instead of immediately setting the perturbed image equal to the re-simulated image, the perturbed image is initially set identical to the original image. Its voxel values are then changed one at a time until all the voxels in the re-simulated domain has been updated. The advantage of doing it this way, is that while changing voxel values one at a time we can update the tree of the original image to be the tree of the new, perturbed image. The Tnew is initially set to be an exact copy of Ttest, and it is then updated iteratively for every voxel that has been assigned a new value. This way of iteratively updating the tree is much cheaper than generating the tree of the perturbed image from scratch.

Once the perturbation of the image and the updating of its tree is completed the frequency distribution can be computed using the tree. This is as usually done by calling Tree2Hist, as this is not an expensive procedure. SimNewImage of course only updates the tree and recomputes the frequency distribution if the perturbed image is in fact different from the original image.

Variable	**Description**	**Type**
Ztest	Current image.	3D integer array

Zcond	Array used to specify which voxels are subject to hard data constraints. The values of such voxels are known and the voxels are used to conditioned upon in the simulation of other voxel values. The simulation algorithm is not allowed to erase and re-simulate the values of voxels that are conditioned upon. Instead these values are considered known and should be used when re-simulating other voxel values.	3D logical array
Zex	This array is the same size as the image Ztest and it is used to keep track of if the pattern that a voxel is the center of exists anywhere in the training image or not. It is used by the procedure getNewIt.	3D logical array
Ttest	Tree of patterns extracted from the image Ztest. The tree of a training image was used to construct Ttest such that it holds only patterns also found in the training image.	tree
Ttrain	Tree of patterns extracted from the training image.	tree
i, j, k	Indices in the image of the voxel that is chosen such that the image is perturbed by erasing and the re-simulating the values of voxels in a domain around it.	integers
nodes	Array with sN rows and three columns. Each row contains the relative coordinates (in the three columns respectively) from a center voxel to one of its neighbouring voxels.	2D integer array
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
Dblock	Contains the parameters that define the domain of an image to be re-simulated to create a perturbed image.	structure
Znew	Perturbed image based on the current image Ztest.	3D integer array

Zexnew	Similar to Zex, this array denotes which voxels in the image Znew are centres of patterns also found in the training image.	3D logical array
Tnew	Tree of patterns extracted from the perturbed image Znew. The tree of a training image was used to construct Tnew such that it holds only patterns also found in the training image.	tree
Hnew	Frequency distribution of the perturbed image Znew.	2D real array
newImage	Indicates whether the perturbed image Znew is different from the original image Ztest (newImage = true) or not (newImage = false).	logical

Appendix A.10. SimVoxel

This procedure is needed to simulate the value of a voxel. It extracts from Ttrain the (unscaled) conditional probability distribution of the value of a center voxel given the values of the neighbouring voxels. It returns this conditional probability distribution and the voxel can then be assigned a value drawn from it. The actual assignment is done by SimNewImage as SimVoxel only returns the unscaled conditional probability density function.

In case the multiple point statistics of Ttrain does not allow for this occurrence of the values of the neighbourhood voxels, i.e., no patterns of the training image matches the partial pattern, the voxels will be dropped one by one from the conditioning until a conditional probability distribution can be extracted. It is always the voxel furthest away from the center voxel that will be dropped. Dropped voxels are assigned the value -1 and they then appear as unknown.

If all neighbouring voxels are dropped the unconditioned distribution of voxel values in the training image will be used as the unscaled probability distribution of voxels also in the image.

Variable	**Description**	**Type**
Zvec	Holds the values of voxels in a neighbourhood. Unknown voxel values are assigned a value of -1.	1D integer array

Ttrain	Tree of patterns extracted from the training image.	tree
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
D	Part of the Dblock structure. Holds the distances from each of the neighbouring voxel to the center voxel.	1D integer array
hc	Holds the unscaled probability distribution of the value of the center voxel conditioned on its neighbouring voxels.	1D real array

Appendix A.11. UpdateSA

The point of this procedure is solely to simplify the code and hopefully make it easier to read by bringing down the number of lines.

The procedure copies an image and its associated variables into another set of variables. This is for instance used in the simulated annealing scheme when a new image is accepted. Then the procedure is used to copy the new image Znew into the variable of the current image Ztest and to update its associated variables such that they contain the variables Tnew, Hnew etc. The updating of the variables is trivial except for the tree which is done by the procedure CopyTree.

Variable	**Description**	**Type**
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
Zin	The image that should be copied.	3D integer array
Zinex	Array to determine which patterns in Zin exist in the training image.	3D logical array
Tin	Tree of the image Zin.	tree

Hin	Frequency distribution of the image Zin.	2D real array
Pin	Array with two elements that holds the value of each term of the objective function of the image Zin.	1D real array
Zout	The image variable into which Zin should be copied.	3D integer array
Zexout	Array to determine which patterns in Zout exist in the training image.	3D logical array
Tout	Variable that holds the tree of the image Zout. This variable should be updated to hold Tin.	tree
Hout	Variable that holds the frequency distribution associated with the image Zout, this variable should be updated to hold Hin.	2D real array
Pout	Variable that holds the values of the terms of the objective function associated with the image Zout, this variable should be updated to hold Pin.	1D real array

Appendix B. Auxiliary Procedures

The following list describes the auxiliary procedures of the FM implementation. These are not of great importance but necessary building blocks.

Appendix B.1. *getNewIt*

This procedure determines which part of an image should be perturbed, i.e., it returns the row, column and layer index of the voxel that is the center of the domain, which should be re-simulated in order to perturb the image. To reduce the number of iterations needed for the simulated annealing algorithm to converge, we wish to choose a voxel that will result in maximal improvement to the perturbed image.

Consider two different voxels in the image. Assume that in an area around the first voxel the image looks very similar to the TI. Erasing the voxel values in a domain around the first voxel and re-simulating them based on the multiple point statistics of the TI will then likely result in a perturbed image that is very similar to the original image. There is then no reason

to expect this perturbed image to be a significantly better solution to the inverse problem, than the image was before perturbing it. Now assume that the original image in an area around the second voxel looks very different than anything that can be seen in the training image. Erasing and re-simulating the voxel values in a domain centred in the second voxel will then create a very different perturbed image. This perturbed image is now much more likely to be a better solution to the inverse problem as it will fit the multiple points statistics of the training image better. And if it is not a better data fit, it could potentially belong to a different, unexplored part of the model space. So at least as long as the image has areas that are dissimilar to any area of the training image, we would like these areas to have a high relative probability to be re-simulated.

The procedure goes through randomly proposed voxels and picks them with a probability that is proportional to the number of undesirable patterns in their neighbourhood. An undesirable pattern is defined as one that does not exist in the training image. This means, the procedure proposes a voxel. It then scans the neighbourhood voxels (say it has y neighbouring voxels) and counts how many of these are centrers of undesirable patterns, let us denote that number x. The voxel is then accepted as a center for the perturbation domain with probability:

$$\text{Prob}(\mathsf{voxel}) \quad = \quad \frac{x}{y+2}$$

The denominator $y + 1$ comes from the number of patterns in the neighbourhood plus the pattern from the voxel itself. The extra +1 is added such that areas with no undesirable patterns, i.e., $x = y$, have a small yet non-zero probability to be chosen. Otherwise the iterative algorithm might be stuck and prevented to converge, as an image can have no undesirable patterns without matching the frequency distribution of the training image and without matching the data fit either.

Variable	**Description**	**Type**

Zex	Denotes which voxels in the image are centres of patterns found also in the training image. An element of Zex is true if the pattern centred in the corresponding voxel exists in the training image. And contrary, an element of Zex is false if the corresponding voxel of the image is center of a pattern not found in the training image.	3D logical array
nodes	Array with sN rows and three columns. Each row contains the relative coordinates (in the three columns respectively) from a center voxel to one of its neighbouring voxels.	2D integer array
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $sN + 1$ voxels.	integer
i, j, k	Indices of the three dimensions of the image for the voxel that is chosen as the center of the domain to be re-simulated.	integers

Appendix B.2. ***getNeighborhood***

Given an image Z and the row, column and layer indices of a voxel in the image, this procedure extracts the voxel values of the neighbouring voxels. The extracted voxel values will be flattened into a 1D array.

Variable	**Description**	**Type**
Z	Image.	3D integer array
i, j, k	Indices in the image of the center voxel of the neighborhood.	integers
nodes	Array with sN rows and three columns. Each row contains the relative coordinates (in the three columns respectively) from a center voxel to one of its neighbouring voxels.	2D integer array
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $sN + 1$ voxels.	integer
Zvec	Holds the values of voxels in a neighbourhood. Unknown voxel values are assigned a value of -1.	1D integer array

innervoxel	Denotes if the voxel with the specified indices was an inner voxel (innervoxel = true) or not (innervoxel = false)	logical

Appendix B.3. *getCPDF*

This is a recursive procedure used by SimVoxel to compute the conditional probability density function of the value of a voxel given the values of its neighbouring voxels.

The procedure goes searching through the tree (depth first) for patterns that matches the neighbourhood values in Zvec and adds up the counts of patterns depending on the value of their center voxel. This way cpdf is a sV+1 element array that has the counts of pattern matching the values of the neighbourhood voxels for each of the sV+1 possible value of the center voxel.

Variable	**Description**	**Type**
Ttrain	Tree of patterns extracted from the training image.	tree
Zvec	Holds the values of voxels in a neighbourhood. Unknown voxel values are assigned a value of -1.	1D integer array
sV	Parameter, the images contain sV+1 categories of voxel values, namely $0, 1, \ldots,$ sV.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have sN+1 voxels.	integer
cpdf	Current counts of patterns matching the values in Zvec.	1D real array

Appendix B.4. *ExtendTree*

This procedure adds another node to a tree. It takes as input a tree node where the pointers in the next array are not associated. The procedure initialises them by allocating their repl arrays and setting the counts to 0. It

also sets their depth values to be one deeper than the current T%depth, and it nullifies their next arrays.

Variable	Description	Type
sV	Parameter, the images contain sV + 1 categories of voxel values, namely $0, 1, \ldots,$ sV.	integer
T	Tree to be extended	tree

Appendix B.5. ShapeTree

This procedure shapes a tree T such that it has the same shape as the Ttrain based on which it was constructed. The original tree cannot be any bigger than the Ttrain, as it holds only patterns that are also found in the training image. But it can be smaller, as some patterns may appear in the training image but not in the image from which the T is constructed. We therefore need to add the nodes representing these patterns but with the count zero.

Ensuring the trees have the same shape simplifies future operations such as comparison of frequency distributions.

Variable	Description	Type
T	Tree of patterns extracted from an image Z	tree
Ttrain	Tree of patterns extracted from the training image.	tree
sV	Parameter, the images contain sV + 1 categories of voxel values, namely $0, 1, \ldots,$ sV.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have sN + 1 voxels.	integer

Appendix B.6. CopyTree

As trees are complex structures we cannot just copy the content of one tree, Told, into another tree, Tnew, in the way we usually do with arrays. Simply saying Tnew = Told has no meaning as it is not defined. This procedure is therefore needed whenever we want to make a copy of a tree. It is used for instance by the UpdateSA procedure to copy the tree of the perturbed image Tnew into the variable holding the tree for the current image Ttest when a perturbed image is accepted.

The procedure initialises a new tree from scratch and then while running through the old tree it copies its content node for node to the new tree without overwriting Told.

Variable	**Description**	**Type**
Told	Tree of patterns extracted from an image.	3D integer array
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
Tnew	Tree identical to Told.	3D integer array

Appendix B.7. DeallocateTree

Recursive procedure that deallocates a tree. This is done by deallocating the repl array and the next array associated with each node for all nodes one node at a time.

Variable	**Description**	**Type**
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
T	Tree node to be deallocated.	tree

Appendix B.8. *UpdateTrainTree*

This recursive procedure is used to add a pattern to the tree of the training image. For each pattern InferTrainTree calls the procedure that then recursively calls itself while going deeper and deeper into the tree. That way the count for the pattern is added all the way to the bottom of the tree. If a type of pattern has not already been added to the tree the procedure ExtendTree is used to extend the tree with extra nodes before the pattern can be added.

Variable	**Description**	**Type**
Ttrain	Current tree node	tree
Zvec	Holds the values of voxels in a neighbourhood.	1D integer array
cv	Voxel value of the center voxel of the pattern.	integer
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer

Appendix B.9. *wrapUpdateTree*

This procedure is called by InferTree to add a pattern to the tree when the center of the pattern is an inner voxel. It works as a wrapper for the recursive UpdateTree.

Variable	**Description**	**Type**
T	Tree of patterns so far extracted from the image Z.	tree
Ttrain	Tree of patterns extracted from the training image.	tree
Zvec	Holds the values of voxels in a neighbourhood.	1D integer array
cv	Voxel value of the center voxel of the pattern.	integer
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer

sN	Parameter, number of neighbours for an inner voxel; which means patterns have sN + 1 voxels.	integer
exist	Indicate whether the pattern is found in the training image (exist = true) or not (exist =false).	logical

Appendix B.10. UpdateTree

Recursive procedure that adds the contribution from a pattern which center voxel is an inner voxel. It goes through the pattern voxel by voxel, and adds it contribution node by node as deep into the tree as allowed. Recall that the shape of the tree must not be changed as it shall remain the same as the shape of the tree of the training image. Therefore patterns from the image that are not found in the training image will not contribute to any counts after a certain level of depth as the nodes representing them do not exist, and they therefore do not appear in the frequency distribution.

The inputs of the procedure is the same as of wrapUpdateTree.

Appendix B.11. wrapUpdateTreeBoundary

Like wrapUpdateTree this procedure is called by InferTree and it works as a wrapper for UpdateTreeBounday. However, this procedure is used to add a pattern which center is not an inner voxel.

Variable	**Description**	**Type**
T	Tree of patterns so far extracted from the image Z.	tree
Ttrain	Tree of patterns extracted from the training image.	tree
cpdfold	Conditional probability distribution of the value of a center voxels conditioned on the values of the neighbouring, potentially imaginary, voxels in Zvec.	1D real array
Zvec	Holds the values of the neighbouring voxels in the pattern. Imaginary voxels have been assigned the value −1.	1D integer array

cv	Voxel value of the center voxel of the pattern.	integer
sV	Parameter, the images contain sV + 1 categories of voxel values, namely $0, 1, \ldots,$ sV.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have sN + 1 voxels.	integer
exist	Indicate whether the pattern is found in the training image (exist = true) or not (exist =false).	logical

Appendix B.12. UpdateTreeBoundary

Recursive procedure that adds the contribution from a pattern which center voxel is not an inner voxel. The contribution of the pattern is set to be the conditional probability of the center value given the voxel values of the neighbouring voxels. This distinction between inner voxels (handled by UpdateTree) and non-inner voxels (handled by UpdateTreeBoundary) is necessary as they contribute differently to the tree.

UpdateTreeBoundary goes through the tree, and for each level it assigns the proper contribution computed based on the counts of patterns in the tree of the training image. It uses the procedure getCPDF to compute the conditional distributions. Like UpdateTree it never changes the shape of the tree as it only add contributions from patterns, that can also be found in the training image.

The inputs of the procedure is the same as of wrapUpdateTreeBoundary.

Appendix B.13. GrowTree

This procedure is called by SimNewImage. It is used to iteratively update the tree, T, of an image, Z, when this is being perturbed. For each changed voxel value upto sN + 1 patterns may have changed and the tree needs to be updated for each of these changes.

GrowTree is called each time a voxel value has been changed, and it makes use of the procedures CenterCount, AddCount, SubtractCount and wrapUpdateTreeBoundary to update the tree. Out of the possibly sN + 1 changed patterns, updating the tree with respect to the pattern of the changed voxel is relatively simple. However, it is done differently depending on whether the

voxel is an inner voxel or not, as this makes it contribute differently to the tree.

The changed voxel might be a neighbour of upto sN voxels, and it therefore might be a part of equally many other patterns. By changing a voxel value these patterns have changed too. The updating of these patterns is a bit tricky and depends on whether or not the changed voxel as well as the voxels of which it is a neighbour are inner voxels or not. Patterns not centred in inner voxels are handled by the same procedure as when the tree was first constructed, namely wrapUpdateTreeBounday.

When pattern changes it might happen that they go from not being a part of the tree to being part of the tree. The way the perturbation of images is done this will often be the case. It might also happen the opposite, namely that patterns used to be in the tree but are not any more. The number of counts in the frequency distribution of the image is for the same reason varying in the different iterations of the simulated annealing algorithm. The procedure is of course able to handle these cases.

Variable	**Description**	**Type**
Z	The image after the voxel value has been changed.	3D integer array
zold	Old value of the changed voxel.	integer
Zex	Array used by getNewIt, should be updated according to the new patterns created by changing the voxel values.	3D logical array
i,j,k	Indices of the changed voxel.	integers
nodes	Array with sN rows and three columns. Each row contains the relative coordinates (in the three columns respectively) from a center voxel to one of its neighbouring voxels.	2D integer array
sV	Parameter, the images contain $\mathsf{sV}+1$ categories of voxel values, namely $0, 1, \ldots, \mathsf{sV}$.	integer
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
Ttrain	Tree of patterns extracted from the training image.	tree

T	Tree of the image before the voxel was changed. The procedures updates it according to the new image Z.	tree

Appendix B.14. CenterCount

This is a recursive procedure called by GrowTree. The procedure is used to update the tree with respect to the pattern centred in the changed voxel when this is an inner voxel. Due to the structure of the tree, the tree is updated by following the (unchanged) voxel values of the neighbourhood voxels and updating the counts of the repl arrays of the corresponding nodes. Say the value of the voxel was changed from i to j then the repl arrays are updated by subtracting 1 count from the i+1th element and adding one count to the j+1th element.

This accounts for the updating for 1 out of the sN+1 patterns possibly affected as explained in the description of GrowTree.

Variable	**Description**	**Type**
T	Tree node to be updated	tree
sN	Parameter, number of neighbours for an inner voxel; which means patterns have sN + 1 voxels.	integer
znew	New value of the changed voxel	integer
zold	Old value of the changed voxel	integer
Zvec	Holds the values of the voxels that are in the neighbourhood of the changed voxel.	1D integer array
exist	Indicate whether the pattern is found in the training image (exist = true) or not (exist =false).	logical

Appendix B.15. AddCount

This is a recursive procedure called by GrowTree. The procedure is used to add a count representing new patterns appearing when the image is per-

turbed. It is used when the voxel, which value was changed, was an inner voxel. This procedure performs the updating of the patterns for those of the sN neighbouring voxels, that are inner voxels.

It loops through those inner voxels, determines which patterns they are now centres of, and adds the count in the tree. Only one value has changed, namely the one belonging to the changed voxel, and the remaining $\mathsf{sN}-1$ values are unchanged. Therefore, if the changed voxel is the ith neighbour in the pattern, then the$i-$!1th first values of the pattern are unchanged and the tree should only be altered from level i and deeper.

Variable	**Description**	**Type**
T	Tree node to be updated	tree
sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
level	Depth level of the tree where the changed voxel will first have effect.	integer
Zvec	Holds the values of the neighbouring voxels in the changed pattern.	1D integer array
zcen	Value of the neighbouring voxel that is center in the changed pattern	integer
exist	Indicate whether the pattern is found in the training image (exist = true) or not (exist =false).	logical

Appendix B.16. SubtractCount

Like AddCount this is a recursive procedure called by GrowTree. Also this procedure is used to update the tree when the image is perturbed. It is used to subtract the count of the old pattern. It starts from the depth of change in voxel values and goes all the way to the bottom of the tree.

Variable	**Description**	**Type**
T	Tree node to be updated	tree

sN	Parameter, number of neighbours for an inner voxel; which means patterns have $\mathsf{sN}+1$ voxels.	integer
level	Depth level of the tree where the changed voxel will first have effect.	integer
Zvec	Holds the values of the neighbouring voxels in the changed pattern.	1D integer array
zcen	Value of the neighbouring voxel that is center in the changed pattern.	integer

Appendix C. Bibliography

Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P., 1983. Optimization by simulated annealing. Science 220 (4598), 671–680.

Lange, K., 2013. Inverse problems in geosciences: Modeling the rock properties of the subsurface of an oil reservoir. Ph.d. dissertation, Technical University of Denmark.

Lange, K., Frydendall, J., Cordua, K. S., Hansen, T. M., Melnikova, Y., Mosegaard, K., 2012. A frequency matching method: Solving inverse problems by use of geologically realistic prior information. Mathematical Geosciences, 1–2110.1007/s11004-012-9417-2.
URL `http://dx.doi.org/10.1007/s11004-012-9417-2`

Strebelle, S., 2002. Conditional simulation of complex geological structures using multiple-point statistics. Mathematical Geology 34, 1–21.

Tarantola, A., 2005. Inverse Problem Theory and Methods for Model Parameter Estimation. SIAM.