MPSlib Documentation

Release 1.5

Thomas Mejer Hansen

Apr 05, 2024

CONTENTS

MPSlib provides a C++ class, and a set of algorithms for simulation of models based on a multiple point statistical (MPS) models inferred from a training image:

- **ENESIM** [GUARDIANO].
- **Generalized ENESIM** [HANSEN2016].
- **Direct Sampling** [MARIETHOZ2010].
- **SNESIM using tree structures** [STREBELLE2002].
- **SNESIM using list structures** [STRAUBHAAR2011].

The above list of algorithms are implemented in two types of algorithms that differ in the way information is inferred from the training image. ENESIM type algorithms samples directly from the training image during simulation, while SNESIM type algorithms scan the training image prior to simulation, and stores the statistics in memory.

- GENESIM:: *mps_genesim*

mps_genesim is an implementation of the generalized ENESIM algorithm [GUARDIANO]. It can run as a pure **ENESIM** algorithm, in which the whole training image is scanned at each iteration, or it can run as a Direct Sampling **DS** algorithm [MARIETHOZ2010], in which the training image is scanned for the first matching event. It can can also run as generalized ENESIM **GENESIM** algorithm in which the training image is scanned for the first N matching event [HANSEN2016].

- SNESIM:: *mps_snesim_tree*/*mps_snesim_list*

Two types of SNESIM type simulation methods are implemented.

mps_snesim_tree stores statistics from the training image in a *tree structure*, as proposed by [STREBELLE2002] . *mps_snesim_list* stores statistics from the training image in a *list structure*, as proposed by [STRAUBHAAR2011].

mps_snesim_tree`and `mps_snesim_list differ only in how the information from the training image is store in memory.

CHAPTER

ONE

CONDITIONAL DATA

All algorithms can handle hard and co-loccated soft data. `mps_genesim` can also handle non-colocated soft data [HANSEN2018].

CHAPTER

TWO

ENTROPY

The entropy of the (unknown) probability distribution related to a specific choice of 1) training image, 2) simulation algorithm, and 3) options for running the simulation algorithm, can optionally be computed as part of simulation. [HANSEN2020].

CHAPTER

THREE

ESTIMATION

All algorithms can optionally be run in *estimation* mode in which the 1D marginal conditional distribution is directly computed (similar to Etype statistics from a number of realizations) [JOHANNSSON2021].

CHAPTER

FOUR

PYTHON AND MATLAB INTERFACE

Interfaces to *Python* and *Matlab/Octave interface* are available.

Python notebooks are a good starting point for using MPSlib with Python.

CHAPTER

FIVE

SOURCE CODE

The latest stable code can be downloaded from http://ergosimulation.github.io/mpslib/.

The current development version is available through GitHub at https://github.com/ergosimulation/mpslib/.

CHAPTER

SIX

BACKGROUND

The goal of developing these codes has been to produce a set of algorithms, based on sequential simulation, for simulation of multiple point statistical models. The code should be easy to compile and extend, and should be allowed for both commercial and non-commercial use.

MPSlib (version 1.0) has been developed by I-GIS and Solid Earth Physics, Niels Bohr Institute.

Development was initially funded by the Danish National Hightech Foundation (now: the Innovation fund) through the ERGO (Effective high-resolution Geological Modeling) project, a collaboration between IGIS, GEUS, and Niels Bohr Institute.

CHAPTER

SEVEN

REFERENCING

Along with the first version of MPSlib a manuscript was published in SoftwareX. Please use this for referencing MPSlib:

> Hansen, T.M., Vu. L.T., and Bach, T. 2016. MPSLIB: A C++ class for sequential simulation of multiple-point statistical models, in *SoftwareX*, doi:10.1016/j.softx.2016.07.001. [pdf,www].

To cite the use of soft data and the preferential path, please use:

> Hansen, T. M., Mosegaard, K., & Cordua, K. S. (2018). Multiple point statistical simulation using uncertain (soft) conditional data. *Computers & geosciences*, 114, 1-10. doi:10.1016/j.cageo.2018.01.017.

To cite the use of MPS based simulation, please use:

> Jóhannsson, Óli D., and Thomas Mejer Hansen (2021). Estimation using multiple-point statistics. *Computers & Geosciences* 156 (2021): 104894. doi:*10.1016/j.cageo.2021.104894 <https://doi.org/10.1016/j.cageo.2021.104894>*

CHAPTER

EIGHT

LICENSE (LGPL)

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

CHAPTER

NINE

THE MANUAL

9.1 Installation and compilation

9.1.1 Download

The latest release, containing statically compiles binaries for Windows and Linux, can be found at https://github.com/ergosimulation/mpslib/releases/latest.

The source can be downloaded from GitHub at https://github.com/ergosimulation/mpslib.

9.1.2 Compilation

The MPSlib codes are written in standard C++11. MPSlib has been developed using the GNU C++ compiler (tested on Windows, Linux and OSX), and Visual Studio C++. Using GNU C++ the code can be compiled using

```
git clone https://github.com/ergosimulation/mpslib.git MPSlib
cd MPSlib
make
```

LINUX (Ubuntu Linux (>16.04))

Prerequisites: g++, which can be installed using

```
sudo apt-get install build-essential
```

Compiler flags:

```
CPPFLAGS = -static -O3 -std=c++11 -Wl,--no-as-needed
```

OSX (XCODE+GCC)

Prerequisites: Xcode, g++

The '-static' option is not available using XCode/OSX, so the following compiler flags are suggested:

```
CPPFLAGS = -O3
```

Windows: (mingw-w64)

MPSlib has been tested using MinGW, specifically mingw-w64 ([http://mingw-w64.org/doku.php]), which can be obtained in several ways. (Note that not all builds of MinGW works!)

One (recommended) approach is to make use of MSYS2. Follow the guide at [http://msys2.github.io/] to install MSYS2, and then install the mingw_w64 toolchain using:

```
pacman -S mingw-w64-x86_64-gcc
pacman -S make
```

Then run "MSYS2 MinGW 64-bit" and/or "MSYS2 MinGW 64-bit" (should present in the windows start menu), and run the 'make' command in the mpslib folder:

```
cd /mnt/c/Users/john/mpslib
make
```

9.2 Running MPS algorithms

The MPS algorithms are run from the commandline using a parameter filename as an argument.
If no argument is given, the default parameter file is assumed to the be name of the simulation algorithm appended with '.txt'.

Therefore

```
mps_genesim
mps_snesim_tree
mps_snesim_list
```

will be equivalent to

```
mps_genesim mps_genesim.txt
mps_snesim_tree mps_snesim_tree.txt
mps_snesim_list mps_snesim_list.txt
```

9.2.1 General options

The following entries appear in all parameter files

Number of realizations

The number of realizations to generate

Random Seed

An integer determines the random seed. A fixed value will return the same realizations for each run.

- [0] assign a 'random' seed at each iteration (new seed every second)

Simulation grid size X, Y, Z

The number of grid cells in the simulation grid

Simulation grid origin X, Y, Z

The value coordinate of the first pixel in the X, Y, and Z direction.

Simulation grid grid cell size X

The size of each pixel in the simulation grid, in the X, Y, and Z direction.

Training image file

The name of the training image file (no spaces allowed).
it must be in GSLIB/EAS ASCII format, and the first line (the 'title') must contain the
dimension of the training file as 'NX NY NZ'.
See the TI folder for examples, and Training image format for more information.

Output folder (spaces in name not allowed)

The path to the folder containing all output. Use forward slash '/' to separate folders.

Shuffle Simulation Grid path (2: preferential, 1: random, 0: sequential) # 2

- [0] sequential path through simulation grid (possibly a multiple grid)
- [1] random path through simulation grid
- [2] preferential path (only useful when soft data is considered)

Entropy factor

When a preferential path is selected the 'EntropyFactor' can be se as a second parameter after the choice of path.

Shuffle Training Image path (1 : random, 0 : sequential)

(Does not affect snesim type algorithms)

- [0] sequential path
- [1] random path

HardData filename

EAS filename with 4 columns: X, Y, Z, and D

HardData search radius

(world units)

Softdata categories

(separated by ;)

Soft datafilenames

(separated by ; only need (number_categories - 1) grids)

Number of threads (minimum 1, maximum 8 - depend on your CPU)

Currently not used.

Debug mode

- [-2]: No information is written to screen or files on disk
- [-1]: + Simulation output is written to files on disk
- [0]: + Information about simulation is written to screen
- [1]: + Simulated realization(s) are shown in terminal
- [2]: + Extra information is written to disk (Random path, number of conditional data, . . .)
- [3]: + Debug information written to screen (results in much slower performance - in general not useful for an end-user)

MASK file

EAS filename with 4 columns: X, Y, Z, and MASK. A mask files of tha same size as the simulation grid can be supplied. '0' in a node indicates a node that will not be simulated. '1' in a node indicates a node that will be simulated.

Entropy

- [0]: No computation of entropy
- [1]: Compute entropy/self-information as part of simulation

See [HANSEN2020] for more information.

Estimation

- [0]: Do not perform sequential estimation
- [1]: Perform sequential estimation (rather than sequential simulation)

See [JOHANNSSON2021] for more information.

9.2.2 GENESIM: Generalized ENESIM

GENESIM (`mps_genesim`) [HANSEN2016] is a generalized version of the ENESIM algorithm [GUARDIANO], in which the conditional distribtion computed from a finite set of conditional events.

In one extreme, the full conditional distribution is obtained by scanning the whole training image at each iteration, in which case GENESIM is identical to the ENESIM algorithm [GUARDIANO].

In another extreme, the conditional distribution is constructed from only one conditional event. In this case GENESIM acts similar to the direct sampling algorithm [MARIETHOZ2010], with the practical difference that the local conditional distribution is in fact computed, and a realization is drawn from. In the direct sampling algorithm the conditional distribution is never realized, instead a new pixel value is chosen from the first matching conditional event.

An example of a parameter file for `mps_genesim`:

```
Number of realizations # 1
Random Seed (0 `random` seed) # 0
Maximum number of counts for conditional pdf # 1
Max number of conditional point # 25
Max number of iterations # 10000
Distance Measure (0: discrete, 1: continious), maximum distance, power # 1 0 0
ColocateDimension # 0
Maximum Search Radius # 1000000
Simulation grid size X # 18
Simulation grid size Y # 16
Simulation grid size Z # 1
Simulation grid world/origin X # 0
Simulation grid world/origin Y # 0
Simulation grid world/origin Z # 0
Simulation grid grid cell size X # 1
Simulation grid grid cell size Y # 1
Simulation grid grid cell size Z # 1
Training image file (spaces not allowed) # ti.dat
```

(continues on next page)

(continued from previous page)

```
Output folder (spaces in name not allowed) # .
Shuffle Simulation Grid path (2: preferential, 1: random, 0: sequential) # 2
Shuffle Training Image path (1 : random, 0 : sequential) # 1
HardData filename  (same size as the simulation grid)# conditional.dat
HardData seach radius (world units) # 1
Softdata categories (separated by ;) # 0;1
Soft datafilenames (separated by ; only need (number_categories - 1) grids) # soft.dat
Number of threads (minimum 1, maximum 8 - depend on your CPU) # 1
Debug mode(2: write to file, 1: show preview, 0: show counters, -1: no ) # -2
```

A description of the options that apply to all MPS algorithms can be seen *here*.

The following lines in the parameter files are specific to the GENESIM type algorithm:

line 3: Maximum number of counts for conditional pdf, `n_max_count_cpdf`

`n_max_count_cpdf` defines the maximum number of counts in the conditional distribution obtained from the training image. When ´n_max_count_cpdf´ has been reached the scanning of the training image stops.

When `n_max_count_cpdf<0` no limit on the number of counts is set.

line 4: Max number for conditional points, `n_cond`

A maximum of `n_cond` conditional data are considered at each iteration when inferring the conditional pdf from the training image.

line 5:Max number of iterations, `n_max_ite`

A maximum of `n_max_ite` iterations of searching through the training image are performed.

if`n_max_ite<0` the full training image is scanned.

line 6: distance_measure, and, `distance_measure`, maximum distance, `distance_max`, and `distance_pow`

The `distance_measure` used:

1: Number of matching pixels (Discrete TI)

2: Euclidean distance (Continuous TI)

The maximum distance what will lead to accepting a conditional template match is set by `distance_max. If not set, is set to distance_max=0`, which means that a perfect match is searched for!

Distance power is used to weight the conditioning data as a function of distance from the center values.
`distance_pow=0` indicated no weighing. A higher will favor the data value of conditional events closer to the center value.
See Mariethoz et al. (2010) Eqn. 2-3. for details.

line 6: 'max_search_radius'

Only conditional data within a radius of 'max_search_radius' is used as conditioning data.

line 7:'colocate_dimension'

For a 3D TI make sure the order matters in the last dimensions (allow performing 2D co-simulation with conditional data in the third dimension)

debug mode

when `debug>1`, A number of extra grids will be written to disk for each realization. If the used training image is called 'ti.dat', then, following GSLIB files contains:

`ti.dat_tg1_0.gslib`: The distance between the conditional event and the corresponding best 'match' in the TI .

`ti.dat_tg2_0.gslib`: The number of matching counts for the conditional pdf.

`ti.dat_tg3_0.gslib`: The index in the TI, of the best matching conditional event.

`ti.dat_path_0.gslib`: Index of the path in the simulation grid.

ENESIM

The classical ENESIM algorithm can be run setting`n_max_count_cpdf` and `n_max_ite` to infinity (using -1):

```
Maximum number of counts for conditional pdf # -1
```

```
Max number of iterations # -1
```

In this case the full training image will be scanned at each iteration to establish a conditional probability density.

ENESIM leads to a very slow algorithm, but the full/most accurate conditional distribtuion is computed at each iteration. This can be usefull when performing simulation conditional to soft data. If not, then the Direct Sampling algorithm is much more efficient (`n_max_count_cpdf=inf`)

GENESIM

In case`0<n_max_count_cpdf<infinity`, `mps_genesim` will behave intermediate between ENESIM and Direct Sampling.

GENESIM is useful in case the local conditional distribution is needed, as is the case when conditioning to soft data. In this case, the GENESIM may be much faster than ENESIM.

DIRECT SAMPLING

In case `n_max_count_cpdf=1`, `mps_genesim` will behave similar to the direct sampling algorithm. The computational efficiency can further be controlled using `n_max_ite,` to be set a value smaller than the number of pixels in the training image.

As the full local conditional distribution is not available (it is never computed/inferred), conditioning to soft data is done using the rejection sampler (Hansen et al. 20xx, submitted)

Temporary Grids

If the verbose level is higher than one 5 temporary grids are written do disk. In case the training image has the name 'ti.dat' the following grids are exported as EAS files :

ti.dat_tg1_0.gslib: The distance for the last accepted match, when scanning the training image.

ti.dat_tg2_0.gslib: The number of counts used to set up the conditional probability density. When using Direct Sampling, `n_max_count_cpdf=1`, this value should never be higher than 1.

ti.dat_tg3_0.gslib: The index of the position in the training image for last/best match.

ti.dat_tg4_0.gslib: The number of iterations in the training image.

ti.dat_tg5_0.gslib: Used number of conditional points.

9.2.3 SNESIM: Single normal equation simulation

The `mps_snesim_tree` and `mps_snesim_list` differ only in the way conditional data are stored in memory (using either a tree like [STREBELLE2002] or a list structure as [STRAUBHAAR2011]).

Both algorithms share the same format for the required parameter file:

```
Number of realizations # 1
Random Seed (0 for not random seed) # 0
Number of mulitple grids # 2
Min Node count (0 if not set any limit) # 0
Max Conditional count (-1 if not using any limit) # -1
Search template size X # 5
Search template size Y # 5
Search template size Z # 1
Simulation grid size X # 100
Simulation grid size Y # 100
Simulation grid size Z # 1
Simulation grid world/origin X # 0
Simulation grid world/origin Y # 0
Simulation grid world/origin Z # 0
Simulation grid grid cell size X # 1
Simulation grid grid cell size Y # 1
Simulation grid grid cell size Z # 1
Training image file (spaces not allowed) # TI/mps_ti.dat
Output folder (spaces in name not allowed) # output/.
Shuffle Simulation Grid path (2: Preferential, 1: random, 0: sequential) # 1
Maximum number of counts for condtitional pdf # 10000
Shuffle Training Image path (1 : random, 0 : sequential) # 1
HardData filaneme  (same size as the simulation grid)# harddata/mps_hard_grid.dat
HardData seach radius (world units) # 15
Softdata categories (separated by ;) # 1;0
Soft datafilenames (separated by ; only need (number_categories - 1) grids) # softdata/
↪mps_soft_xyzd_grid.dat
Number of threads (minimum 1, maximum 8 - depend on your CPU) # 1
Debug mode(2: write to file, 1: show preview, 0: show counters, -1: no ) # 1
```

A few lines in the parameter files are specific to the SNESIM type algorithms, and will be discussed below:

line 3, `n_mul_grids`

`n_mul_grids` defines the number of multiple grids used. `n_mul_grids=0`, means that no multiple grid will be used.

line 4, `n_min_node`

The search tree will only be searched to a level where the number of counts in the conditional distribution is above `n_min_node`.

line 5, `n_cond`

`n_cond` is the maximum number of conditional point used, within the search template

lines 6-8, the search template, `tem_nx`, `tem_ny`, `tem_nz`

The search template defines the size of the template that is used to prescan the training image
and store (using a tree or list) the conditional distribution for all configurations of the data template.

Varying template size [only valid for mps_snesim_tree]

Optionally a the template size can vary for different multiple grid sizes. The first number refer to the template size at the coarsest multiple grid. The last number refer to the template size at the finest grid (simulated last). The template size for intermediate grid sizes is found by linear interpolation, and output to the screen if the debug mode is above 0. The use of varying template sizes can reduce computation time considerable, with only little effect on the pattern reproduction.

For example the following defines a 9x9x1 template at the coarsest grid, and a 3x3x1 grid at the finest grid

```
Search template size X # 9 3
Search template size Y # 9 3
Search template size Z # 1 1
```

Also,

```
Search template size X # 5 5
Search template size Y # 5 5
Search template size Z # 5 5
```

is equivalent to

```
Search template size X # 5
Search template size Y # 5
Search template size Z # 5
```

9.3 Training image format

Training images must be formatted as EAS/GSLIB ascii file, with the special requirement that that the first line must describe the dimensions of the training image

```
2 3 1
1
Channel
                  1
                  1
                  0
                  0
                  1
                  1
```

Line 1: Contains the dimension of training image, specified as 3 integeres seprated by a space `nx ny nz`

Line 2: The number columns of data (typically one for a training image)

Line 3: The name of column 1, [There should be one line, with the name of each column, for each number of columns]

Line4 -> : The training image data. One data value on each of the following `nx*ny*nz` lines.

9.4 Examples

The section contains different example of the use of MPSlib.

9.4.1 Ex: Varying template size in SNESIM

mps_snesim_tree and *mps_snesim_list* allowing using a template size that changes for each multiple grid. The template size is given as a value for the coarsest multiple grid, and a value for the final dense simulation grid. Linear interpolation is used to compute the template size at each multiple grid

For example using a template size of 8x7x4 oon the coarsest grid and a template size of 4x3x3 on the finest grid can be given in the mps_snesim parameter file as

```
Search template size X # 8 4
Search template size Y # 7 3
Search template size Z # 4 3
...
```

The main reason for using variable template size is that, typically, a considerable amount of CPU is used in the finer simulation grids to prune (remove) conditional data.

The computational speed and effect on simulation results can be investigated by simulating with a fixed random seed for a number of different choices of template (from mpslib_snesim_varying_template.py). Figure Fig. 9.1 shows one realization obtained using a fixed template size of 11x11x1 (upper left) compared to using varyaing templates. The startingf template (used at the coarse multiple grid) is set to 11x11x1 in all cases. The template at the finest grid is tested for 10x10x1, 9x9x1, ..., 1x1x1. The lower left figure shows the simulation time for each realization. Figure Fig. 9.2 show the computational speedup relative to using a fixed full size template.

The computational speedup is more significant doing 3D simulation.

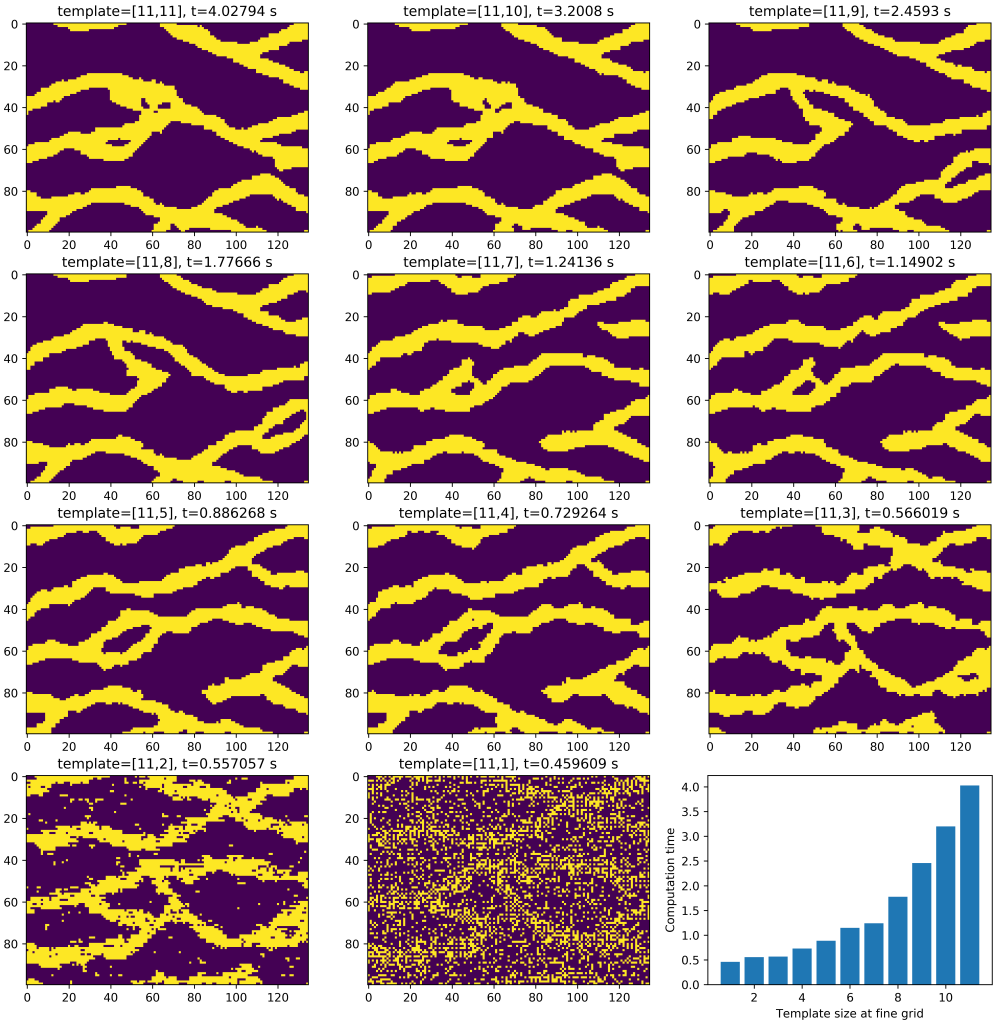


Fig. 9.1: E-type mean using a sequential, random and preferential simulation path, conditioning to 3 non-co-located soft data.

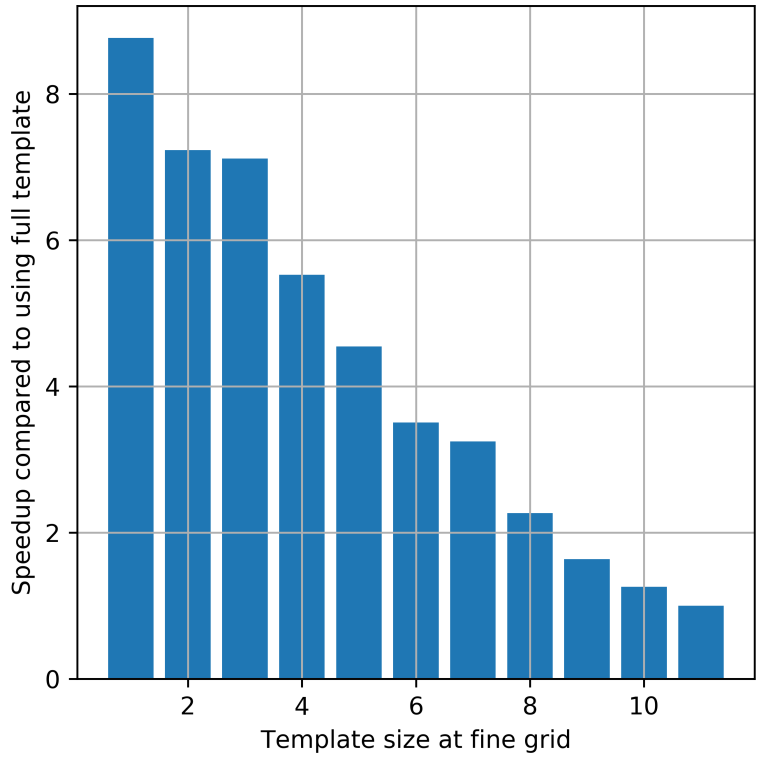


Fig. 9.2: CPU speedup compared to using a fixed template of size 11x11x1.

9.4.2 Ex: Soft/uncertain data

MPSlib can take 'soft' data into account. 'soft' data is defined as uncertain and spatially independent information about one ore more model parameters. Formally the soft information is quantified by $f_{soft}(\mathbf{m})$ as

$$\begin{aligned} f_{soft}(\mathbf{m}) &\sim f_{soft}(m_1, m_2, m_3, ..., m_M) \\ &= f_{soft}(m_1)\ f_{soft}(m_2)\ ...\ f_{soft}(m_M) \\ &= \prod_i^M f_{soft}(m_i) \end{aligned} \tag{9.1}$$

The assumption of spatial independence is critical. If the uncertain information is in fact spatially dependent (as is typically the case using soft data derived from inversion of geophysical data), the variability in the generated realizations will be too small, such the apparent information content is too high.

MPSlib allow conditioning to both co-located soft data (mps_snesim_tree and mps_genesim) and non-co-located soft data (mps_genesim). The implementation of soft in MPSlib is described in detail in in [HANSEN2018].

Soft data must be provided as an EAS file. If a training image with *Ncat* categories is used then the EAS file must contain N=3+`Ncat` columns. The first three must be ´X´, *Y*, and *Z*. The the following columns provide the probability of each category. Column 4 (the first column with soft data) refer to the probability of the category with the lowest number in the training image.

An example of defining 3 soft data, for a case with *Ncat=2*, and with soft information close to hard information (almost no uncertainty) is

```
SOFT data mimicking hard data
5
X
Y
Z
P(cat=0)
P(cat=1)
        6          14          0       0.001       0.999
        13         16          0       0.001       0.999
        3          14          0       0.999       0.001
```

Co-located soft data

The usual approach to handling soft data, is to conisder on co-located soft data during sequential simulation. This means that at each iteration of sequential simulation one sample from

$$f(m_i|I_{hard}, I_{soft}) = f_{TI}(m_i|\mathbf{m}_c) * f_{soft}(m_i)$$

As demonstrated in [HANSEN2018] the use of a unilateral or random path using co-located soft data leads to ignoring most of the soft information. The problem is most severe when using scattered soft data. If in stead a simulation path is chosen where more informed nodes (where the entropy of the soft data i high) are visited preferentially to less informed nodes, then much more of the soft data is being taken into account.

The default path in MPSlib is therefore the *preferential* path, that can selected as the path type *2* in the parameter file. The second parameter controls the randomness of the preferential path.

```
...
Training image file (spaces not allowed) # ti.dat
Output folder (spaces in name not allowed) # .
```

(continues on next page)

(continued from previous page)

```
Shuffle Simulation Grid path (2: preferential, 1: random, 0: sequential, 2:␣
↪preferential) # 2 4
Shuffle Training Image path (1 : random, 0 : sequential) # 1
...
```

The behavior of *mps_genesim* with soft data is controlled by the number of soft conditional data, and the max search radius of conditional soft data. To use co-located soft data, the number of soft data is set to 1, and the search radius is set to 0 as :

```
...
Max number of conditional point: Nhard, Nsoft# 16 1
...
Max Search Radius for conditional data [hard,soft] # 10000000 0
...
```

Figure Fig. 9.3 shows the point wise mean of 100 realizations using the soft data described above, in case using a sequential, random and preferential simulation path (from mpslib_hard_as_soft_data.py): .

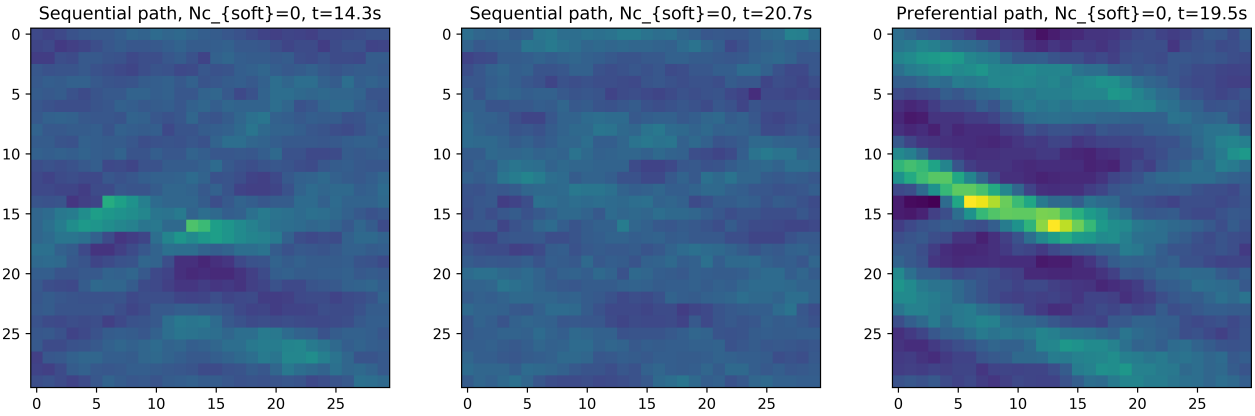


Fig. 9.3: E-type mean using a sequential, random and preferential simulation path, conditioning co-located soft data.

and

Non Co-located soft data

If soft information is scattered, and located relatively far away from each other, then using only co-located soft data my work well. But, when soft information is more densely available, using only co-located soft data results in disregarding available information.

mps_genesim can handle non-colocated soft information running both in ENESIM mode and Direct Sampling mode (using only 1 match in the training image). In both cases one samples from the following conditional distribution during sequential simulation:

$$f(m_i|I_{hard}, I_{soft}) = f_{TI}(m_i|\mathbf{m}_c) * \prod_{j=1}^{Nc_{soft}} f_{soft}(m_j)$$

where Nc_{soft} refer to the number of (the closest) soft conditional points to use. This number of defined right next to the maximum number of hard data used for condisioning. In order to use non-co-located soft data, the search radius for soft data must be set to a value larger than 0. In the example below, the closest 25 hard and 3 soft data is used:

```
:linenos:
:lineno-start: 1
:emphasize-lines: 4

Number of realizations # 1
Random Seed (0 `random` seed) # 1
Maximum number of counts for conditional pdf # 1
Max number of conditional point: Nhard, Nsoft# 25 3
Max number of iterations # 1000000
...
Max Search Radius for conditional data [hard,soft] # 10000000 10000000
...
```

Figure Fig. 9.4 shows the point wise mean of 100 realizations using a sequential, random and preferential simulation path (from mpslib_hard_as_soft_data.py) using two non-colocated soft data.

Note how the sequential and random path can in principle be used, as part of the soft data is used at each iteration, but that the simulation time is dramatically higher than using the preferential path (10 to 20 times faster). The speed is us due to the simulation of the nodes of the soft data the start of the simulation. When the soft data has been simulated, the will in effect be treated as previously simulated hard data, and hence the simulation will perform as normal conditional sequential simulation.

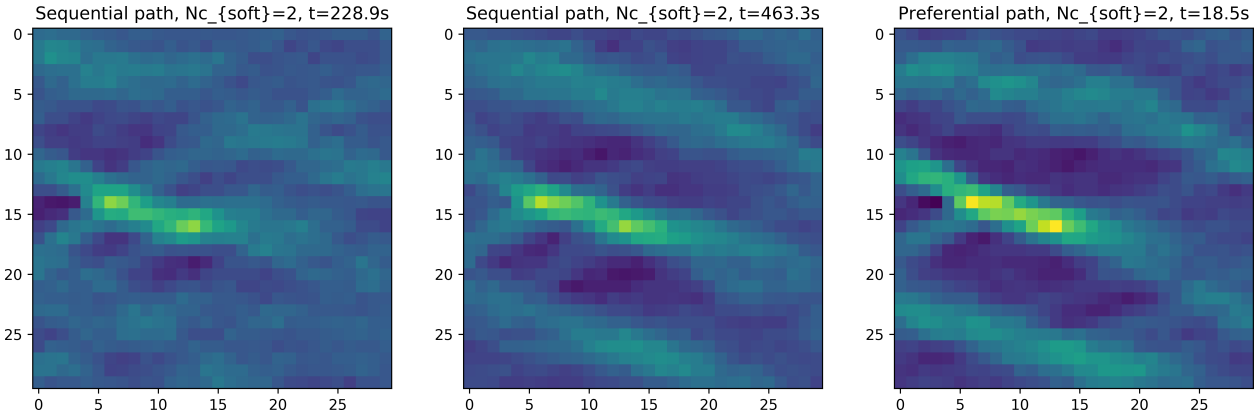


Fig. 9.4: E-type mean using a sequential, random and preferential simulation path, conditioning to 3 non-co-located soft data.

9.5 Matlab interface

A simple Matlab interface to the algorithms on MPSlib has been developed. It consists of the following m-files:

Running MPSlib algorithms:

- mps_cpp.m: Run MPSlib algorithms
- mps_cpp_thread.m: Split MPSlib simulation on multiple threads
- mps_cpp_clean.m: Clean up files after running MPSlib.

Reading and writing parameter files:

- mps_snesim_read_par.m
- mps_snesim_write_par.m

- mps_enesim_read_par.m
- mps_enesim_write_par.m

Examples:

- mps_cpp_example.m
- mps_cpp_example_softwarex.m
- mps_cpp_example_estimation.m
- mps_cpp_example_entropy.m

These m-files requires no special toolboxes, and are compatible with GNU Octave.

`mps_cpp` takes to three inputs, of which the first two are mandatory:

```
TI  : [1D/2D/3D] matrix with a training image
SIM : [1D/2D/3D] simulation grid of NaN values
O   : Object controlling the simulation. (optional)
```

`mps_cpp.m` can be used to perform MPS simulation using both `mps_genesim`, `mps_snesim_tree`, and `mps_snesim_list`. By default `mps_snesim_tree` is used unless the choice of simulation algorithm is set in the `O.method` field:

```
O.method='mps_snesim_tree';
O.method='mps_snesim_list';
O.method='mps_genesim';
```

9.5.1 Getting started in Matlab

The simplest approach to using `mps_cpp` is to use for example

```
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulation grid
[reals,O]=mps_cpp(TI,SIM);
```

This will use the classical channel based training image (from Strebelle (2000)), and perform unconditinoal simulation (using `mps_snesim_tree`) in 2D grid of size 80x60 pixels. `reals` will contain one single generated realization, and the O structure will be populated with all the parameters used for `mps_snesim_tree`:

```
O =

struct with fields:

                          null: ''
                          debug: -1
                          rseed: 0
                  output_folder: '.'
                      WriteTI: 1
                  ti_filename: 'ti.dat'
          simulation_grid_size: [60 80 1]
                      origin: [0 0 0]
              grid_cell_size: [1 1 1]
                      mask_filename: 'mask.dat'
          hard_data_filename: 'd_hard.dat'
```

(continues on next page)

(continued from previous page)

```
                        method: 'mps_snesim_tree'
            parameter_filename: 'mps_snesim.txt'
                        n_real: 1
                n_multiple_grids: 3
                n_min_node_count: 0
                        n_cond: 39
                    template_size: [5 5 1]
        shuffle_simulation_grid: 2
    entropyfactor_simulation_grid: 4
                shuffle_ti_grid: 1
        hard_data_search_radius: 1
            soft_data_categories: '0;1'
            soft_data_filename: 'soft.dat'
                        n_threads: 1
                    doEstimation: 0
                        doEntropy: 0
                    exe_filename: 'F:\PROGRAMMING\mpslib\matlab\..\mps_snesim_tree.exe'
                            time: 0.2945
                                x: [1×60 double]
                                y: [1×80 double]
                                z: 0
                            clean: 1
```

9.5.2 SNESIM type simulation

SNESIM, using both search trees and list for lookup, is available using both `mps_snesim_tree``and ``mps_snesim_list`. Both algorithms make use of the same parameters (and parameter file)'. The choice of simulation algortihm is done using:

```
O.method='mps_snesim_list';
O.method='mps_snesim_tree';
```

The main parameters specific for `mps_snesim_tree` and `mps_snesim_list` are

```
n_multiple_grids: 3  # Number of multiple grids
n_min_node_count: 0  # min number of counts in conditional pdf
        n_cond: 39   # number of conditional data
    template_size: [5 5 1]  # the templated size
```

A dynamic template size canbe set using

```
O.template_size = [15 15 1; 5 5 1]';
```

that suggests a template size of [15 15 1] is used at the coarse grid, and [5 5 1] at the finest grid.

9.5.3 GENESIM type simulation

A simple GENESIM type simulation can be obtained using

```
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulationgrid
O.method='mps_genesim';
[reals,O]=mps_cpp(TI,SIM,O);
```

which return the Odata structure:

```
O =

struct with fields:

                        method: 'mps_genesim'
                            debug: -1
                            rseed: 0
                    output_folder: '.'
                        WriteTI: 1
                    ti_filename: 'ti.dat'
            simulation_grid_size: [60 80 1]
                        origin: [0 0 0]
                grid_cell_size: [1 1 1]
                    mask_filename: 'mask.dat'
            hard_data_filename: 'd_hard.dat'
            parameter_filename: 'mps_genesim.txt'
                        n_real: 1
                        n_cond: [25 1]
                        n_max_ite: 1000000
                n_max_cpdf_count: 1
        shuffle_simulation_grid: 2
    entropyfactor_simulation_grid: 4
                shuffle_ti_grid: 1
        hard_data_search_radius: 100000
            soft_data_categories: '0;1'
            soft_data_filename: 'soft.dat'
                        n_threads: 1
                distance_measure: 1
                    distance_min: 0
                    distance_pow: 0
            colocated_dimension: 0
                max_search_radius: [1000000 1000000]
                    doEstimation: 0
                        doEntropy: 0
                    exe_filename: 'F:\PROGRAMMING\mpslib\matlab\..\mps_genesim.exe'
                            time: 1.9083
                                x: [1×60 double]
                                y: [1×80 double]
                                z: 0
                        clean: 1
```

The main parameters specific for `mps_genesim` are

```
          n_cond: [25 1]     % maximum number of conditional data for
                             % hard and soft data
       n_max_ite: 1000000    % maximum number of iteration in the ti
n_max_cpdf_count: 10         % maximum counts for the conditional pdf
```

The distance `measure_measure`, `measure_min`, `measure_pow` controls hwo the distance is computed for discrete and continious parameters:

```
distance_measure: 1
    distance_min: 0
    distance_pow: 0
```

GENESIM as ENESIM

`mps_genesim` can act as a classical ENESIM algorithm by scanning the whole training image at each iteration:

```
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulationgrid
O.method='mps_genesim';
O.n_max_ite=1e+9 ; Iterate 'forever'
O.n_max_cpdf_count=1e+9 % No upper limit on number of counts for conditional pdf
[reals,O]=mps_cpp(TI,SIM,O);
```

GENESIM as DIRECT SAMPLING

`mps_genesim` can act as the DIRECT SAMPLING algorithm by scanning whole training image only until one (the first) matching event is found, i.e. by at each iteration:

```
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulationgrid
O.method='mps_genesim';
O.n_max_ite  = 1000
O.n_max_cpdf_count=1 ; % No upper limit on number of counts for conditional pdf
[reals,O]=mps_cpp(TI,SIM,O);
```

GENESIM, a hybrid between ENESIM and DIRECT SAMPLING

GENESIM can run as a hybrid between DIRETC SAMPLING and ENESIM, by setting `n_max_cpdf_count` somewhere between 1 (DIRECT SAMPLING) and infinitty (ENESIM). This is especially usefule when conditioning to soft data-

```
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulationgrid
O.method='mps_genesim';
O.n_max_ite  = 1000
O.n_max_cpdf_count=10 ; %
[reals,O]=mps_cpp(TI,SIM,O);
```

9.5.4 Plot simulation results

`mps_cpp_plot`, can be used used to plot simulation results

```
[reals,O]=mps_cpp(TI,SIM,O);
mps_plot_cpp(reals,O);
```

If debug level is larger than one, then the number of temporary grids with different information, is also visualized.

```
O.debug_level=2;
[reals,O]=mps_cpp(TI,SIM,O);
mps_plot_cpp(reals,O);
```

9.5.5 Parallel simulation

When simulating more than one realization, `mps_cpp_thread` can be used to split the simulation onto several threads, such that simulation will be performed in parallel. (This requires Matlab with the Matlab Parallel toolbox)

```
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulation grid
O.method='mps_snesim_tree';
O.n_real=10;

% simulation on one CPU
t0=now;
[reals]=mps_cpp(TI,SIM,O);
disp(sprintf('Elapsed time (sequential): %g s',(now-t0)*(3600*24)))

% simulation on multiple CPUs (require the Matlab Parallel toolbox)
t0=now;
[reals]=mps_cpp_thread(TI,SIM,O);
disp(sprintf('Elapsed time (parallel): %g s',(now-t0)*(3600*24)))
```

Provides the following output, running on 4 threads:

```
Elapsed time (sequential): 21.326 s
mps_cpp_thread: Using 4 threads/workers
mps_cpp_thread: running thread #4 in mps_04
mps_cpp_thread: running thread #3 in mps_03
mps_cpp_thread: running thread #2 in mps_02
mps_cpp_thread: running thread #1 in mps_01
Elapsed time (parallel): 6.835 s
```

9.5.6 Sequential Estimation

All of `mps_genesim`, `mps_snesim_tree`, `mps_snesim_list` can used to perform conditinoal 'estimation', rather the the default sequential simulation, simply by setting `O.doEstimation=1`.

Details about using sequential estimation with MPS algorithms can be found in [JOHANNSSON2021]

```
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulationgrid
SIM(10:12,20)=0; % some conditional data
SIM(40:40:43)=1; % some conditional data
O.method='mps_genesim';
O.doEstimation=1;

[reals,O]=mps_cpp(TI,SIM,O);
est = O.cg; % this of size [80,60,2] as the training image has 2 soft_data_categories
```

Sequential estimation can be performed in parallel, consideiring each pixel at a time. This is utilised in `mps_cpp_estimation` that use parallel threads for faster estimation:

```
O.n_max_cpdf_count=100000;
[est]=mps_cpp_estimation(TI,SIM,O);
```

9.5.7 Self-information and Entropy

The self-information for realizations can be computed by setting `O.doEntropy=1`. Details about computing the self-information is found in [HANSEN2020].

In this case the self-information of each realization is returned in `O.SI`, and the entropy is the simply the average of `O.SI`.

```
clear all;
TI=mps_ti;           %  training image
SIM=zeros(80,60).*NaN; %  simulation grid
O.method='mps_snesim_tree';
O.doEntropy=1;
O.n_real = 10;
[reals,O]=mps_cpp(TI,SIM,O);

% The self-information of each realizations is
O.SI =

      431.6090
      364.8060
      415.4050
      378.6850
      425.6930
      402.5930
      524.6750
      475.0100
      336.9290
      489.7420
```

(continues on next page)

(continued from previous page)

```
% Compute the entropy as the average self-information
H_est = mean(O.SI)

      H_est =

  424.5147
```

9.6 scikit-mps: a Python interface to MPSlib

scikit-mps

`scikit-mps` [https://pypi.org/project/scikit-mps/] is a Python module that interfaces to MPSlib.

It can be installe using pip by (see details at https://pypi.org/project/scikit-mps/):

```
pip install scikit-mps
```

Or from the source code located in the `mpslib/scikit-mps` folder, using

```
cd mpslib/scikit-mps
pip install .
```

To allow editing scikit-mps locally after install use

```
cd mpslib/scikit-mps
pip install -e .
```

Several Python notebooks examples are located in `mpslib/scikit-mps/examples` whic can be browsed here: *Python notebooks*.

It includes 4 submodules

```
mps.mpslib
mps.eas
mps.trainingimages
mps.plot
```

and

`mps.mpslib` contains the core function for setting up and running the algorithms in MPSlib. `mps.eas` contains function to read and write EAS formatted ASCII files. `mps.trainingimages` provides easy access to 2D and 3D training images. `mps.plot` provides 2D/3D plotting utilities.

It makes use of matplotlib (for 2D graphics) and pyvista [http://docs.pyvista.org/] (for 3D graphics).

A simple example of using scikit-mps to generate 4 realizations using `mps_snesim_tree` is (from mpslib_simple.py):

```
import mpslib as mps

# Initialize the MPS object, using a specific algorithm (def='mps_snesim_tree')
O=mps.mpslib(method='mps_snesim_tree')

# Select number of realiization [def=1]
O.par['n_real']=4

# Set training image
O.ti = mps.trainingimages.strebelle()[0]
O.plot_ti()

# Run MPSlib
O.run()

# Plot the results
O.plot_reals()

O.plot_etype()
```

that provides Figures Fig. 9.5 and Fig. 9.6.

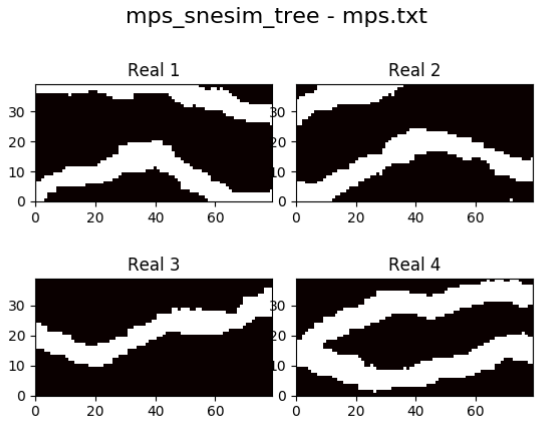


Fig. 9.5: Realizations from simulation

and

9.6.1 mps.mpslib: The main interface to MPSlib

`mps.mpslib` provide a class to allow running MPSlib algorithms. An instance `O` of the class is class can be created using:

```
O=mps.mpslib()
```

This will use a default choice of simulation method, as defined in `O.method`

```
In [1]: O.method
Out[1]: 'mps_genesim'
```

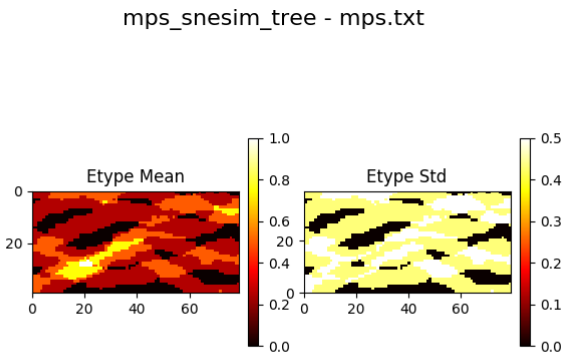


Fig. 9.6: Etype mean and variance from the simulation.

and a default parameter file name, as defined in `O.parameter_filename`

```
In [2]: O.parameter_filename
Out[2]: 'mps.txt'
```

and default parameters for the parameter file, as defined in `O.par`

```
In [3]: O.par
Out[3]:
{'n_real': 1,
'rseed': 1,
'n_max_cpdf_count': 1,
'out_folder': '.',
'ti_fnam': 'ti.dat',
'simulation_grid_size': array([80, 40,  1]),
'origin': array([0., 0., 0.]),
'grid_cell_size': array([1, 1, 1]),
'mask_fnam': 'mask.dat',
'hard_data_fnam': 'hard.dat',
'shuffle_simulation_grid': 2,
'entropyfactor_simulation_grid': 4,
'shuffle_ti_grid': 1,
'hard_data_search_radius': 1,
'soft_data_categories': array([0, 1]),
'soft_data_fnam': 'soft.dat',
'n_threads': 1,
'debug_level': -1,
'n_cond': 36,
'n_cond_soft': 0,
'n_max_ite': 1000000,
'distance_measure': 1,
'distance_min': 0,
'distance_pow': 1,
'colocate_dimension': 0,
'max_search_radius': 10000000,
```

(continues on next page)

(continued from previous page)

```
'max_search_radius_soft': 10000000}
```

All these parameters can be set when the object is initialized. A common approach to initialized the mpslib object is to initialize it using a specific choice a simulation algorithm, simulation grid size, number of realizations, and number of conditional points. This can be done using e.g.

```
O = mps.mpslib(method='mps_snesim_tree',
    simulation_grid_size(80,80,1),
    n_cond = 49
    n_real = 1)
```

To run the MPSlib algorithms using a single thread use:

```
O.run()
```

To run the MPSlib algorithms using a multiple threads use:

```
O.run_parallel()
```

9.6.2 mps.eas: reading and writing EAS formatted files

mps.eas contains several functions for reading and writing EAS formatted data.

Read EAS point set

To read a point data set, use

```
import mpslib as mps
EAS = mps.eas.read('data.dat')
```

`EAS['D']` contains the data values [ndata X ncolumns] as a 2D numpy array. `EAS['header']` contains the header for each columns as list of strings `EAS['title']` contains the title [string] for the eas file.

Write EAS point set

To write a matrix as an EAS formatted point set

```
import mpslib as mps
mps.eas.write(D, filename='eas.dat', title='eas title', header=[]):
```

`D` must be a 2D numpy array. `filename` is the EAS file name. Optionally `header` [list of strings] and `title` [string] can be set.

Read EAS volume set

An EAS volume data set, is a special version of the EAS file format that allow describing a 1D-3D volume. The first line (the title) must contain the dimensions of the data in the eas file formatted as e.g.

```
100 210 13
```

to describe a matrix of size nx=100, ny=210, and nz=13. It is read as for the points data set

```
import mpslib as mps
EAS = mps.eas.read('ti.dat')
```

`EAS['Dmat']` contains a 3D numpy array of shape (100,210,13)

Write EAS volume set

A 3D numpy array can be written as an EAS volume set using

```
import mpslib as mps
import numpy as no
D = np.zeros((20,10,30))
mps.eas.write_mat(D,filename='D,dat')
```

9.6.3 mps.trainingimages: Easy access to training images.

mps.traningimages contain easy access to a large number of training images. To see a list of the available training images call

```
In [40]: mps.trainingimages.ti_list()
Available training images:
checkerboard - 2D checkerboard
checkerboard2 - 2D checkerboard - alternative
strebelle - 2D discrete channels from Strebelle
lines - 2D discrete lines
stones - 2D continious stones
bangladesh - 2D discrete Bangladesh
maze - 2D discrete maze
rot90 - 3D rotation 90
rot20 - 3D rotation 20
horizons - 3D continious horizons
fluvsim - 3D discrete fluvsim
```

To load and plot the widely used training image from Strebelle, simply call it using e.g.

```
import mpslib as mps
ti, ti_filename = mps.trainingimages.strebelle()
mps.plot.plot_3d(ti)
```

To load and plot a checkerboard training image, use e.g

```
import mpslib as mps
ti, ti_filename = mps.trainingimages.checkerboard()
mps.plot.plot_3d(ti)
```

To load and plot the 3D fluvsim training image, use e.g

```
import mpslib as mps
ti, ti_filename = mps.trainingimages.fluvsim()
mps.plot.plot_3d(ti)
```

9.6.4 mps.plot: Plotting utilities

‘’mps.plot” contains a number of functions for plotting mpslib data and realizations in 2D (using matplotlib) and 3D (using pyvista).

plot_reals_3d()

To plot several realizations using pyvista from a mpslib object, use

```
import mpslib as mps
O = mps.mpslib(n_real=4)
O.run
O.plot.plot_reals_3d(O)
```

To plot a 3D numpy array using pyvista use

```
import mpslib as mps
ti, ti_filename = mps.trainingimages.checkerboard()
mps.plot.plot_3d(ti)
```

To slice the 3D grid use

```
import mpslib as mps
ti, ti_filename = mps.trainingimages.checkerboard()
mps.plot.plot_3d(ti, slice=1)
```

To plot only valúes in a specific range, e.g. -.5 to 0.5, use

```
import mpslib as mps
ti, ti_filename = mps.trainingimages.checkerboard()
mps.plot.plot_3d(ti, threshold = (-0.5, 0.5))
```

9.7 Pyhthon Notebook examples

Perhaps the easiest to get starting using MPSlib is by running and adjusting a number of jupyter notebooks.

9.7.1 MPSlib: Getting started with MPSlib/scikit-mps in Python

This a small example getting started with MPSlib through an iPython notebook

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import mpslib as mps
```

Setup MPSLib

First one need to initialize an instance of the mpslib object.

[2]:
```
# Initialize MPSlib using default algortihm, and seetings
O = mps.mpslib();

# Initialize MPSlib using the mps_snesim_tree algorthm, and a simulation grid of size␣
↪[80,70,1]
O = mps.mpslib(method='mps_snesim_tree', simulation_grid_size=[80,70,1])

# specific parameters can be parsed directly when calling mps.mpslib (as abobve), or set␣
↪by updating the O and O.par structure as
#O.parameter_filename = 'mps_snesim.txt'
O.par['debug_level']=-1
O.par['n_cond']=25
O.par['n_real']=16
O.par['n_threads']=5
O.par['do_entropy']=1
O.par['simulation_grid_size']=np.array([80,50,1])
```

```
Using mps_genesim installed in /mnt/f/PROGRAMMING/mpslib/scikit-mps/mpslib/bin (scikit-
↪mps in /mnt/f/PROGRAMMING/mpslib/scikit-mps/mpslib/mpslib.py)
Using mps_snesim_tree installed in /mnt/f/PROGRAMMING/mpslib/scikit-mps/mpslib/bin␣
↪(scikit-mps in /mnt/f/PROGRAMMING/mpslib/scikit-mps/mpslib/mpslib.py)
```

[3]:
```
# All adjustable parameters for the specifric chosen MPSlib algorithm are
O.par
```

[3]:
```
{'n_real': 16,
 'rseed': 1,
 'n_max_cpdf_count': 1,
 'out_folder': '.',
 'ti_fnam': 'ti.dat',
 'simulation_grid_size': array([80, 50,  1]),
 'origin': array([0., 0., 0.]),
 'grid_cell_size': array([1, 1, 1]),
 'mask_fnam': 'mask.dat',
 'hard_data_fnam': 'hard.dat',
 'shuffle_simulation_grid': 2,
 'entropyfactor_simulation_grid': 4,
 'shuffle_ti_grid': 1,
 'hard_data_search_radius': 1,
 'soft_data_categories': array([0, 1]),
```

(continues on next page)

(continued from previous page)

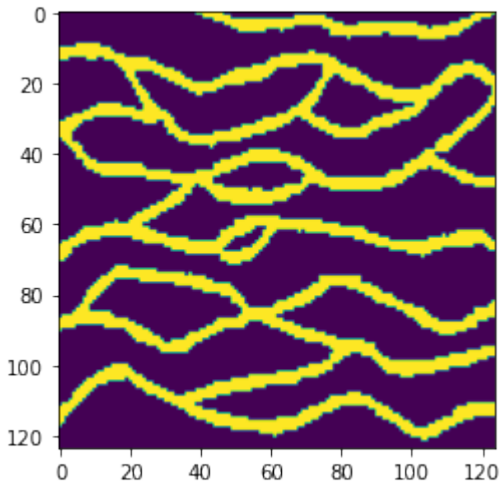
```
 'soft_data_fnam': 'soft.dat',
 'n_threads': 5,
 'debug_level': -1,
 'do_estimation': 0,
 'do_entropy': 1,
 'template_size': array([8, 7, 1]),
 'n_multiple_grids': 3,
 'n_min_node_count': 0,
 'n_cond': 25}
```

Choose training image

[4]:
```
TI, TI_filename = mps.trainingimages.strebelle(di=2, coarse3d=1)
#TI, TI_filename = mps.trainingimages.rot90()
O.par['ti_fnam']=TI_filename
plt.imshow(TI[:,:,0].T)
```

```
Beginning download of https://github.com/GAIA-UNIL/trainingimages/raw/master/MPS_book_
↪data/Part2/ti_strebelle.sgems to ti_strebelle.dat
```

[4]:
```
<matplotlib.image.AxesImage at 0x7f8ce3029bb0>
```



Run MPSlib

The chosen MPSlib algorithm is run using a single thread by executing

```
O.run()
```

and using multiple threads by executing

```
O.run_parallel()
```

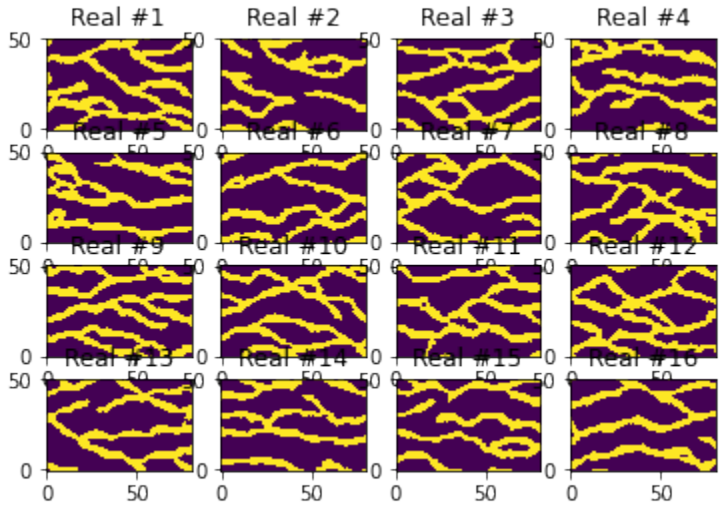
[5]:
```
#O.run()
O.run_parallel()
```

```
parallel: Using 4 of max 10 threads
```

[5]:
```
[<mpslib.mpslib.mpslib at 0x7f8ce0c1aeb0>,
 <mpslib.mpslib.mpslib at 0x7f8ce319f580>,
 <mpslib.mpslib.mpslib at 0x7f8ce0c1ae80>,
 <mpslib.mpslib.mpslib at 0x7f8ce319f6a0>]
```

Plot some realizations using matplotlib

[6]:
```
O.plot_reals()
```



[ ]:

9.7.2 MPSlib: hard and soft data in MPSlib

MPSlib can account for hard and soft data (both colocated and no-colocated). Detail about the use of the preferential path and co- and non-co-located soft data can be found in

Hansen, Thomas Mejer, Klaus Mosegaard, and Knud Skou Cordua. "Multiple point statistical simulation using uncertain (soft) conditional data." Computers & geosciences 114 (2018): 1-10

`mps_snesim_tree` and `mps_snesim_list` can account for both colocated soft data only.

`mps_genesim`can account for both colocated and non-colocated soft data.

Define hard data

Hard data (model parameyers with no uncertainty) are given by the ‘”d_hard”’ variable, with X, Y, Z, and VALUE for each conditonal data. 3 conditional hard data can be given by

```
O.d_hard = np.array(  [[ ix1, iy1, iz1, val1],
                       [ ix2, iy2, iz2, val2],
                       [ ix3, iy3, iz3, val3]])
```

Define soft/uncertain data

Soft data (model parametrs wth no uncertainty) are given by the ‘”d_soft variable, with X, Y, Z, for the position, and a probability of each possible outcome. When considering a training with two categories [0,1], then with P(m=0)=0.2, at position [5,3,2] can be set as

```
O.d_soft = np.array(  [[ 5, 3, 2, 0.2 0.8]])
```

If a training image has 3 categories and P(m=0)=0.2, and P(m=1)=0.3, then

```
O.d_soft = np.array(  [[ 5, 3, 2, 0.2, 0.4, 0.5]])
```

Preferential path

MPSlib makes use of a preferential simulation path, such that model parameters with more informed conditional information (i.e. wigh lower entropy) prior to less nodes with less informed conditonal information. Especially when using spare soft data, the use of a preferential path should be preferred

```
O.par['shuffle_simulation_grid']=0 # Unilateral path
O.par['shuffle_simulation_grid']=1 # Random path
O.par['shuffle_simulation_grid']=2 # Preferential path
```

co-located soft data

By default only co-located soft data are considered during simulation, as given by

```
O.par['n_cond_soft']=1 # only 1 soft data is used
O.par['max_search_radius_soft'] = 0 # only co-located soft data us used
O.par['shuffle_simulation_grid']= 2 # Preferential path
```

Whenever using only co-locate soft data is is adviced to use the preferential path

non-co-located soft data.

Even when using the preferential path, model parameters with informed conditional indformation, close to the point being simulated, will not be taken into account. This means in practice that not all information in soft conditional data is used. As an alternative 'mps_genesim' can handle non-colocated soft data, by using a rejection sample to accept a proposed match m* from scannining the TI, with a probability proprtional to the product of the condtional information evalauted in m*.

This means that one can account for, in principle, any number of soft data, as one can account for any number of hard data. In practice, it becomes computationally har to account for many soft data. To set the number of soft data used for conditining to 3, on can use

```
O.par['n_cond_soft']=3
O.par['max_search_radius_soft'] = 10000000 # only co-located soft data us used
O.par['shuffle_simulation_grid']=2  # Preferential path
```

When using multiple (or all) conditional soft data, then use of the preferential path may not lead to more informed realizations than using a random path, but simulation mey be sginificantly faster using the preferential path as model parameters with soft data will be simulated first, and the subsequenty simulation will be conditional to only hard data, and hence computationally more efficient. Therefore it is advised to use a preferential path always.

[1]:
```
import mpslib as mps
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
#O=mps.mpslib(method='mps_snesim_tree', parameter_filename='mps_snesim.txt')
O=mps.mpslib(method='mps_genesim', parameter_filename='mps_genesim.txt')

TI1, TI_filename1 = mps.trainingimages.strebelle(3, coarse3d=1)
O.par['soft_data_categories']=np.array([0,1])
O.ti=TI1
```

```
Using mps_genesim installed in /mnt/space/space_au11687/PROGRAMMING/mpslib (scikit-mps␣
↪in /mnt/space/space_au11687/PROGRAMMING/mpslib/scikit-mps/mpslib/mpslib.py)
```

[3]:
```
O.par['rseed']=1
O.par['n_multiple_grids']=0;
O.par['n_cond']=16
O.par['n_cond_soft']=1
O.par['n_real']=500
O.par['debug_level']=-1
O.par['simulation_grid_size'][0]=18
O.par['simulation_grid_size'][1]=13
O.par['simulation_grid_size'][2]=1
O.par['hard_data_fnam']='hard.dat'
O.par['soft_data_fnam']='soft.dat'
O.delete_local_files()

O.par['n_max_cpdf_count']=100
```

Hard data

[4]:
```
# Set hard data
d_hard = np.array([[ 15, 4, 0, 1],
                   [ 15, 5, 0, 1]])
# Optionally use hard data
# O.d_hard = d_hard
```

Soft/uncertain data

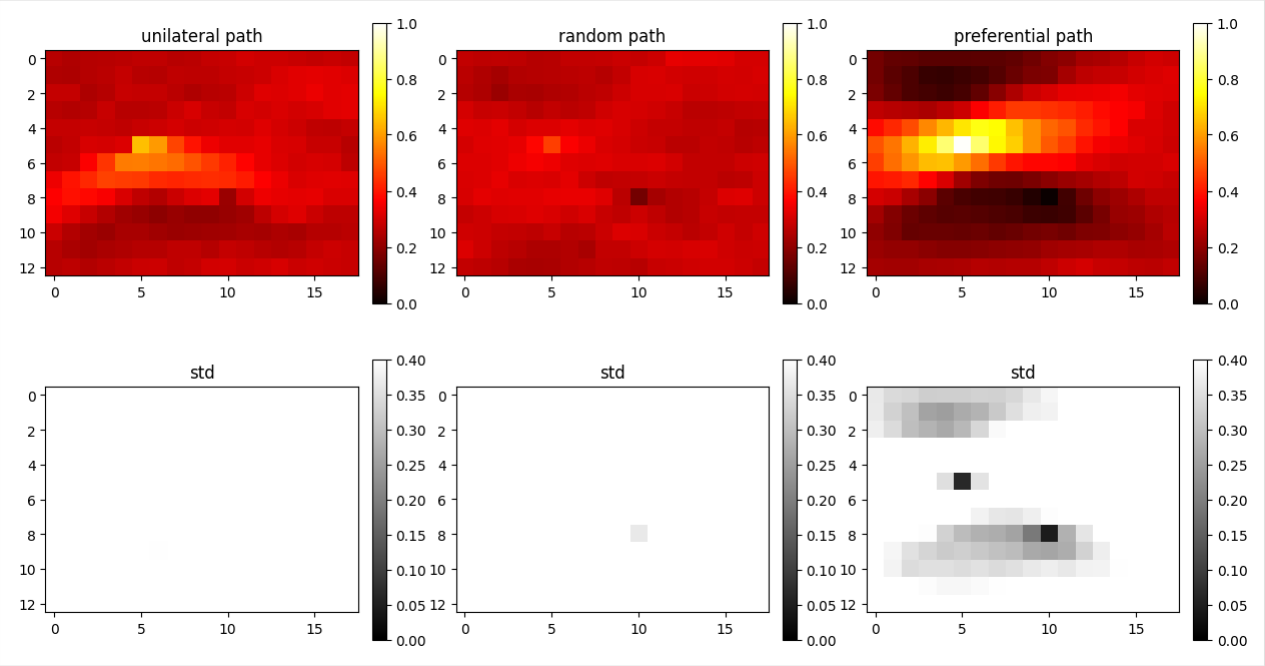
[5]:
```
# Set soft data
d_soft = np.array([[ 2, 2, 0, 0.7, 0.3],
                   [ 5, 5, 0, 0.001, 0.999],
                   [ 10, 8, 0, 0.999, 0.001]])

O.d_soft = d_soft
```

Example 1: co-locational soft data only

In this example only one soft data is used, and only if it located a the same location as being simulated in the sequential simulation method.

[6]:
```

# Only co-locational
O.par['n_cond_soft']=1
O.par['max_search_radius_soft'] = 0

gtxt=['unilateral','random','preferential']
shuffle_simulation_grid_arr = [0,1,2]
fig = plt.figure(figsize=(15, 8))
for i in range(len(shuffle_simulation_grid_arr)):
    # Set preferential path
    O.par['shuffle_simulation_grid']=shuffle_simulation_grid_arr[i]

    O.delete_local_files()
    O.run_parallel()
    m_mean, m_std, m_mode=O.etype()

    plt.subplot(2,3,i+1)
    plt.imshow(m_mean.T, zorder=-1, vmin=0, vmax=1, cmap='hot')
    plt.colorbar(fraction=0.046, pad=0.04)
    plt.title('%s path' % gtxt[i])
    plt.subplot(2,3,3+i+1)
    plt.imshow(m_std.T, zorder=-1, vmin=0, vmax=0.4, cmap='gray')
    plt.title('std')
    plt.colorbar(fraction=0.046, pad=0.04)
```

```
parallel: Using 25 of max 26 threads
parallel: Using 25 of max 26 threads
parallel: Using 25 of max 26 threads
```



Example 2: One non-co-locational soft data

In this example still only one soft data is used, however, it can be located at any location in tyhe simulation grid.

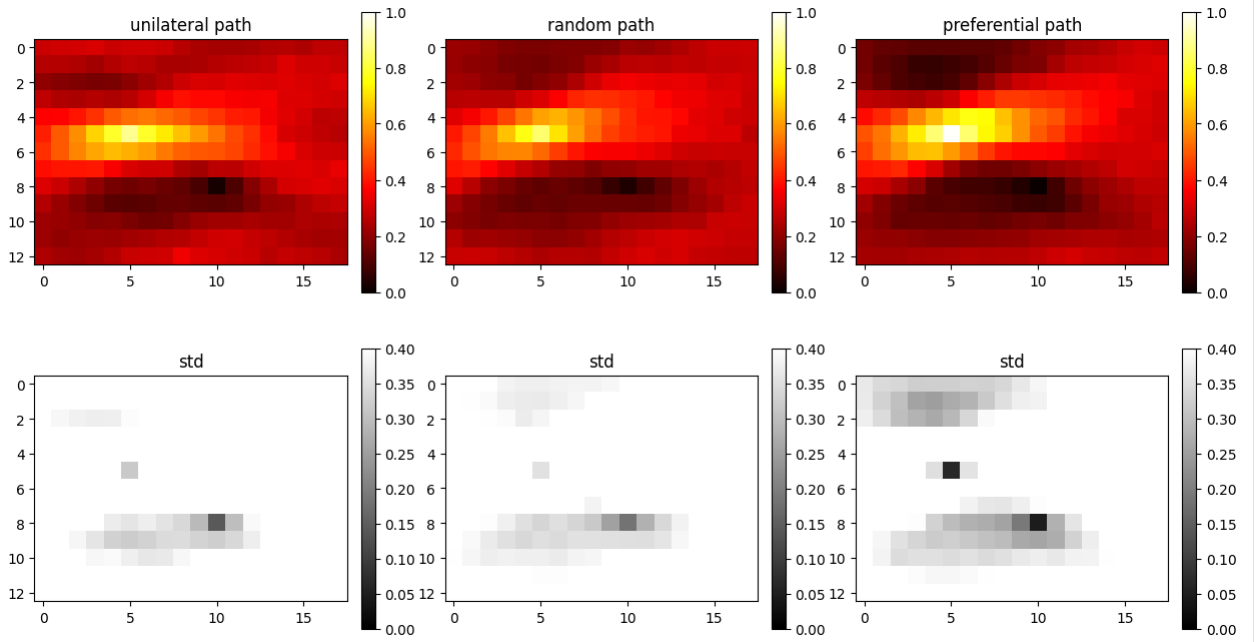
[7]:
```
# Only 1 co-locational
O.par['n_cond_soft']=1
O.par['max_search_radius_soft'] = 1000000

shuffle_simulation_grid_arr = [0,1,2]
fig = plt.figure(figsize=(15, 8))
for i in range(len(shuffle_simulation_grid_arr)):
    # Set preferential path
    O.par['shuffle_simulation_grid']=shuffle_simulation_grid_arr[i]

    O.delete_local_files()
    O.run_parallel()
    m_mean, m_std, m_mode=O.etype()

    plt.subplot(2,3,i+1)
    plt.imshow(m_mean.T, zorder=-1, vmin=0, vmax=1, cmap='hot')
    plt.colorbar(fraction=0.046, pad=0.04)
    plt.title('%s path' % gtxt[i])
    plt.subplot(2,3,3+i+1)
    plt.imshow(m_std.T, zorder=-1, vmin=0, vmax=0.4, cmap='gray')
    plt.title('std')
    plt.colorbar(fraction=0.046, pad=0.04)
```

```
parallel: Using 25 of max 26 threads
parallel: Using 25 of max 26 threads
```

(continues on next page)

(continued from previous page)

```
parallel: Using 25 of max 26 threads
```



Example 3: 3 (all) non-co-locational soft data

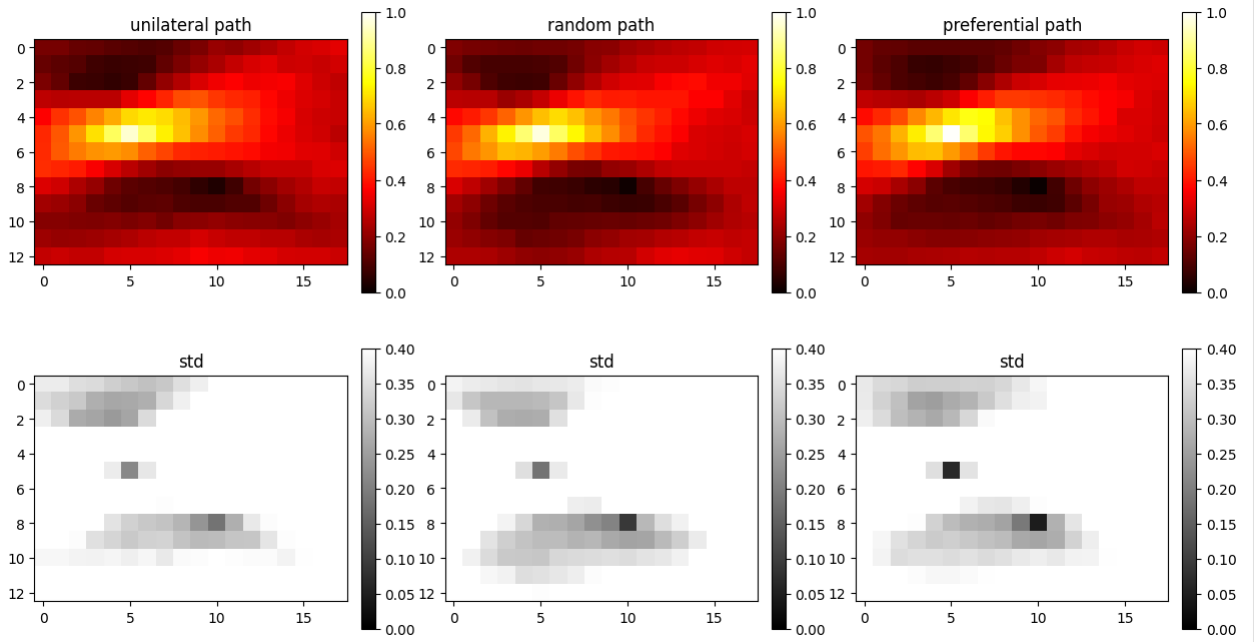
[8]:
```
# Three co-locational
O.par['n_cond_soft']=3
O.par['max_search_radius_soft'] = 1000000

shuffle_simulation_grid_arr = [0,1,2]
fig = plt.figure(figsize=(15, 8))
for i in range(len(shuffle_simulation_grid_arr)):
    # Set preferential path
    O.par['shuffle_simulation_grid']=shuffle_simulation_grid_arr[i]

    O.delete_local_files()
    O.run_parallel()
    m_mean, m_std, m_mode=O.etype()

    plt.subplot(2,3,i+1)
    plt.imshow(m_mean.T, zorder=-1, vmin=0, vmax=1, cmap='hot')
    plt.colorbar(fraction=0.046, pad=0.04)
    plt.title('%s path' % gtxt[i])
    plt.subplot(2,3,3+i+1)
    plt.imshow(m_std.T, zorder=-1, vmin=0, vmax=0.4, cmap='gray')
    plt.title('std')
    plt.colorbar(fraction=0.046, pad=0.04)
```

```
parallel: Using 25 of max 26 threads
parallel: Using 25 of max 26 threads
```

(continues on next page)

(continued from previous page)

```
parallel: Using 25 of max 26 threads
```

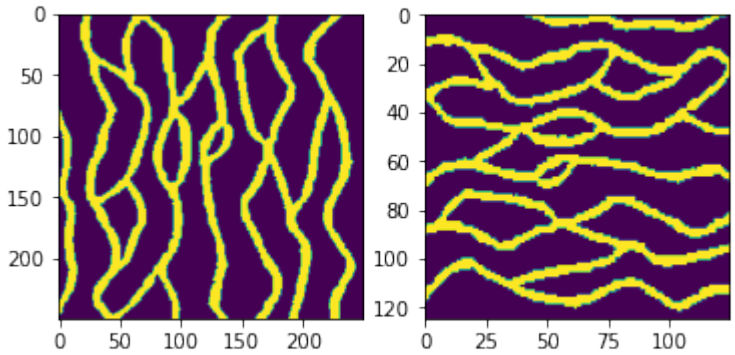


[ ]:

9.7.3 MPSlib: Using masks

[1]:
```
import mpslib as mps
import matplotlib.pyplot as plt
import numpy as np
import copy
from numpy import squeeze
```

[2]:
```
# TI1: Strebelle
TI1, TI_filename1 = mps.trainingimages.strebelle(di=1)
TI1=np.swapaxes(TI1,0,1)
# TI1: Strebelle, rotated and coarsened
TI2, TI_filename2 = mps.trainingimages.strebelle(di=2)
plt.figure(1)
plt.subplot(1,2,1)
plt.imshow(np.transpose(TI1[:,:,0]))
plt.subplot(1,2,2)
plt.imshow(np.transpose(TI2[:,:,0]))
plt.show()
```

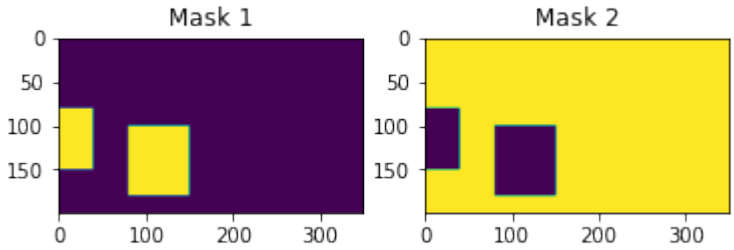


[3]:
```
#%% MPS_SNESIM_TREE
grid_size = np.array([350, 200, 1])
O = mps.mpslib(method='mps_snesim_tree',
                    n_real = 1, verbose_level=-1)
#O = mps.mpslib(method='mps_genesim',
#                    n_real = 1, verbose_level=-1)
O.par['debug_level']=-1
O.par['n_cond']=49
O.par['simulation_grid_size']=grid_size

# make sure no unwanted hard/soft data files are being used
O.delete_local_files()
```

[4]:
```
d_mask1=np.zeros([grid_size[0],grid_size[1],grid_size[2]])
d_mask1[80:150,100:180]=1;
d_mask1[0:40,80:150,]=1;
d_mask2=1-d_mask1;
mask_fnam1='mask_01.dat'
mask_fnam2='mask_02.dat'
mps.eas.write_mat(d_mask1,mask_fnam1)
mps.eas.write_mat(d_mask2,mask_fnam2)
```

[4]:
```
{'dim': {'nx': 350, 'ny': 200, 'nz': 1},
 'n_cols': 1,
 'title': '350 200 1',
 'header': ['Header'],
 'D': array([1., 1., 1., ..., 1., 1., 1.])}
```

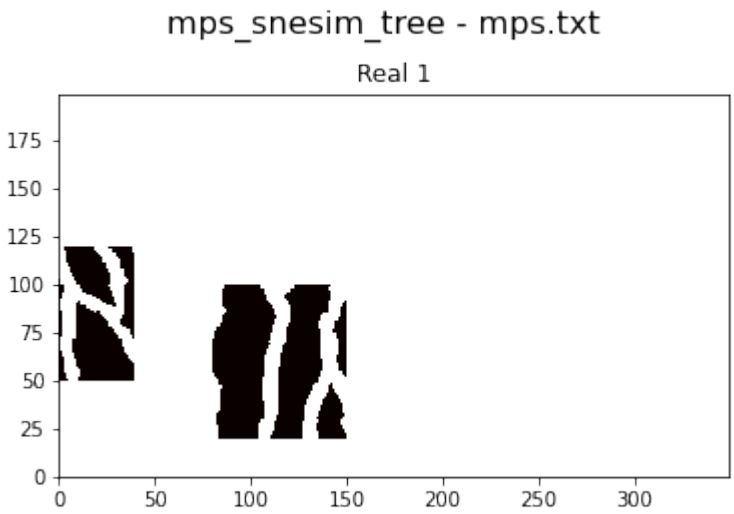
[5]:
```
plt.figure(2)
plt.subplot(121)
plt.imshow(np.transpose(np.squeeze(d_mask1)))
plt.title('Mask 1')
plt.subplot(122)
plt.imshow(np.transpose(np.squeeze(d_mask2)))
plt.title('Mask 2')
plt.show()
```



[6]:
```
#%% Simulation in region/mask 1
O1=copy.deepcopy(O)
O1.delete_hard_data()
O1.par['mask_fnam']=mask_fnam1;
#O1.par['ti_fnam']=TI_filename1;
O1.ti=TI1
O1.run()
```

[6]:
```
True
```

[7]:
```
plt.figure(3)
O1.plot_reals()
```



[8]:
```
d_hard = O1.hard_data_from_sim()
```

```
Number of non-nan data: 8400
```

[9]:
```
d_hard = O1.hard_data_from_sim()
#%% SImulation in region/mask 2
O2=copy.deepcopy(O)
O2.parameter_filename='mps_mask2.par'
O2.par['mask_fnam']=mask_fnam2;
O2.ti=TI2
#O2.par['ti_fnam']=TI_filename2;
O2.delete_hard_data()
```

(continues on next page)

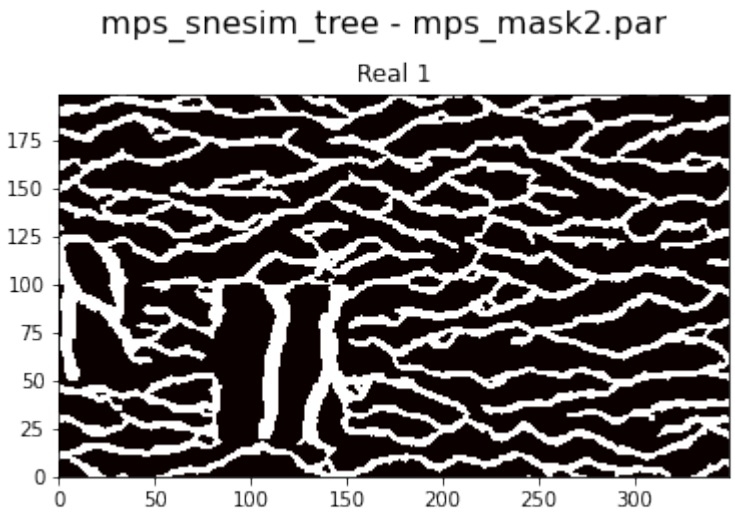
(continued from previous page)

```
O2.d_hard = d_hard
O2.run()
```

```
Number of non-nan data: 8400
```

[9]:
```
True
```

[10]:
```
plt.figure(4)
O2.plot_reals()
```



[ ]:

[ ]:

9.7.4 MPSlib: Training images in scikit-mps

scikit-mps conrtains a module, mpslib.trainingimages, that allow loading and creating a number of training images

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import mpslib as mps
import pyvista
pyvista.set_plot_theme("document")
pyvista.global_theme.jupyter_backend = 'panel'  # use this in jupyter lab
#pyvista.global_theme.jupyter_backend = 'pythreejs' # use this in notebook
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

List the available training images

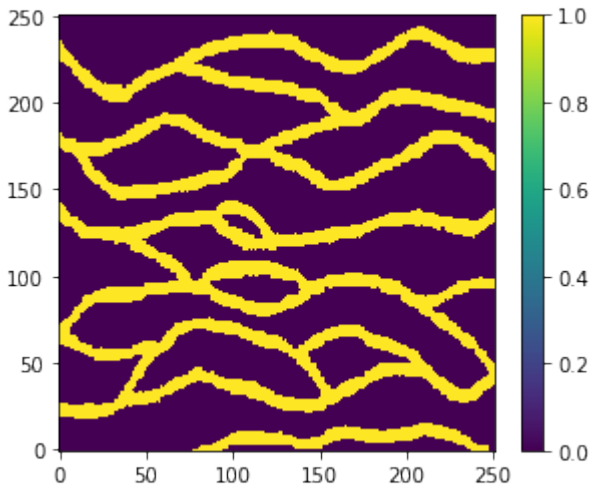
[2]:
```
TI_type, TI_desc = mps.trainingimages.ti_list()
```

```
Available training images:
   checkerboard - 2D checkerboard
  checkerboard2 - 2D checkerboard - alternative
      strebelle - 2D discrete channels from Strebelle
          lines - 2D discrete lines
         stones - 2D continious stones
     bangladesh - 2D discrete Bangladesh
           maze - 2D discrete maze
          rot90 - 3D rotation 90
       horizons - 3D continious horizons
        fluvsim - 3D discrete fluvsim
```

Plot training images

Load and plot a single training image

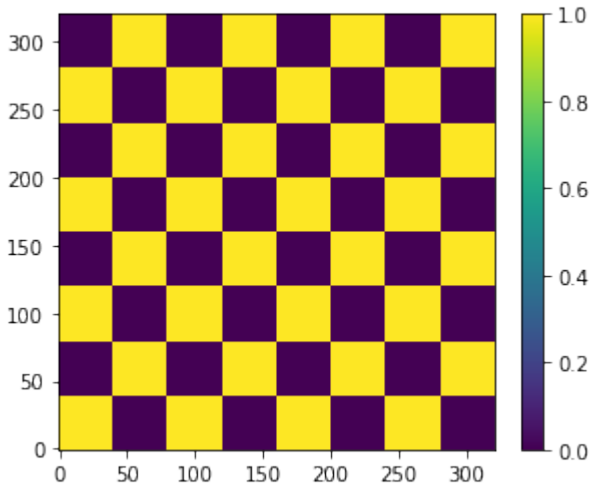
[3]:
```
TI, TI_filename=mps.trainingimages.strebelle()
TI.shape
mps.plot.plot_2d(TI, header='Strebelle')
```



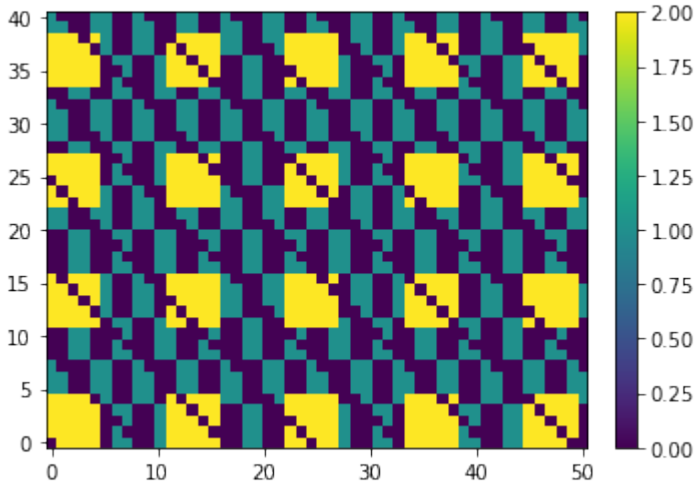
Plot all training images

[4]:
```
for i in range(len(TI_type)):
    ti_type=TI_type[i]
    print('Loading and plotting %s' % (ti_type) )
    O_TI=getattr(mps.trainingimages,ti_type)
    TI, TI_fname=O_TI()
    mps.plot.plot(np.squeeze(TI), slice=0, header=TI_desc[i])
```

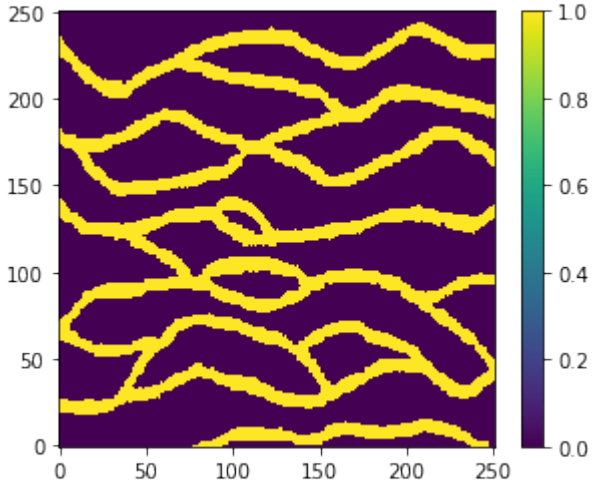
```
Loading and plotting checkerboard
```



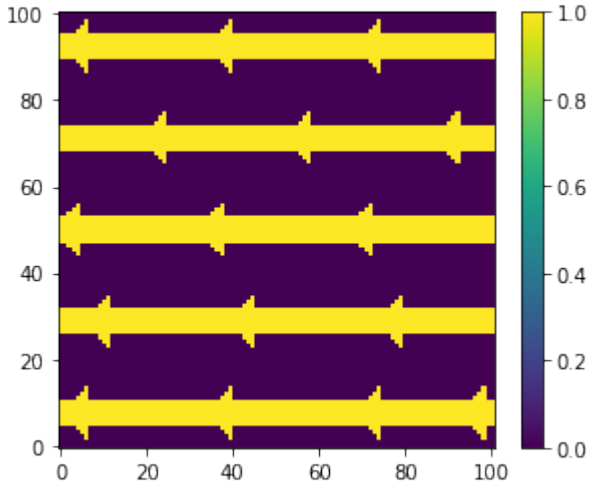

```
Loading and plotting checkerboard2
```



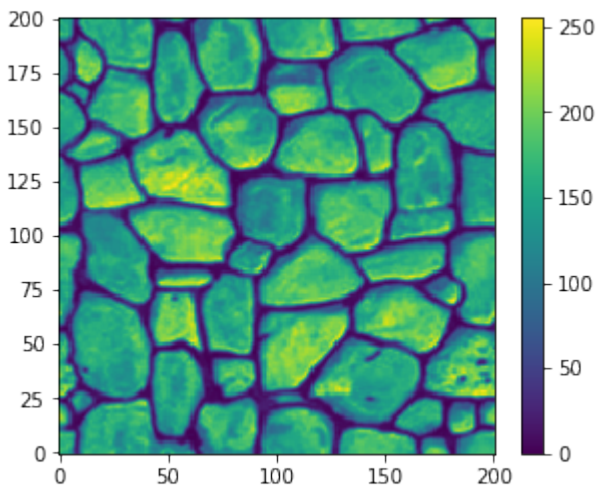

```
Loading and plotting strebelle
```



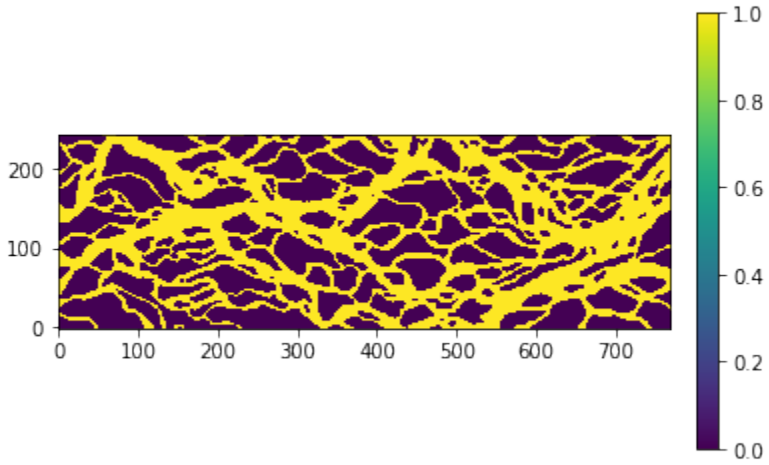
```
Loading and plotting lines
Beginning download of https://github.com/GAIA-UNIL/trainingimages/raw/master/MPS_book_
↪data/Part2/ti_lines_arrows.sgems to ti_lines.dat
```



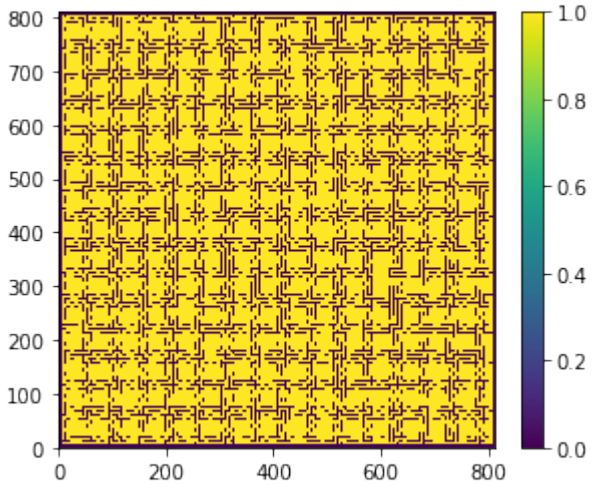

```
Loading and plotting stones
Beginning download of https://github.com/GAIA-UNIL/trainingimages/raw/master/MPS_book_
↪data/Part2/ti_stonewall.sgems to ti_stones.dat
```




```
Loading and plotting bangladesh
Beginning download of https://github.com/GAIA-UNIL/trainingimages/raw/master/MPS_book_
↪data/Part2/bangladesh.sgems to ti_bangladesh.dat
```




```
Loading and plotting maze
Beginning download of https://raw.githubusercontent.com/cultpenguin/mGstat/master/ti/
↪maze.gslib to ti_maze.dat
```




```
Loading and plotting rot90
Beginning download of https://github.com/GAIA-UNIL/trainingimages/raw/master/MPS_book_
↪data/Part2/checker_rtoinvariant_90.zip to ti_tot90.dat.zip
Unziping ti_tot90.dat.zip to ti_tot90.dat
```

```
/home/tmeha/miniconda3/envs/mps/lib/python3.9/site-packages/pyvista/core/dataset.py:1555:
↪ PyvistaDeprecationWarning: Use of `cell_arrays` is deprecated. Use `cell_data`␣
↪instead.
  warnings.warn(
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

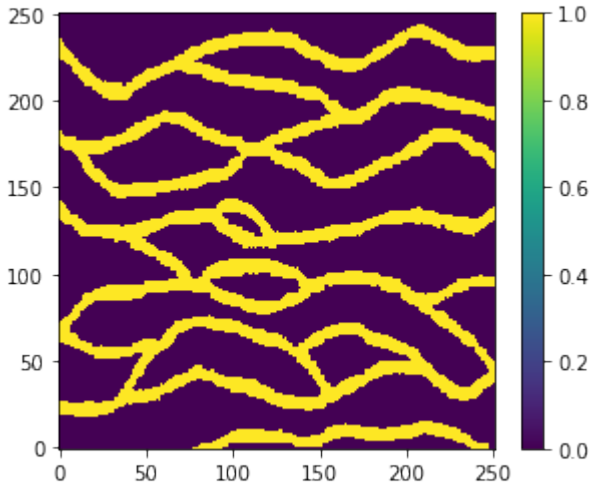
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

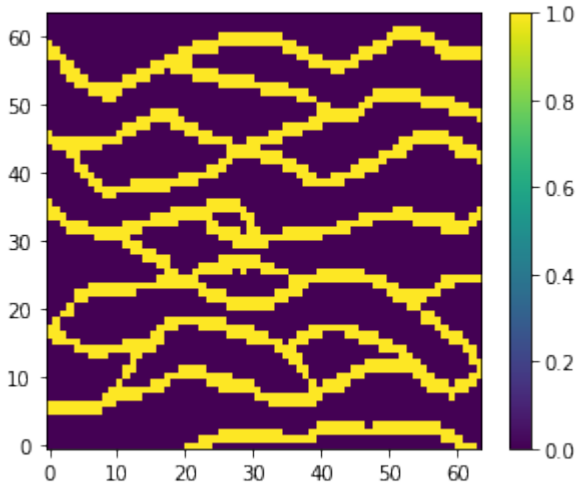
```
Loading and plotting horizons
Beginning download of https://github.com/GAIA-UNIL/trainingimages/raw/master/MPS_book_
↪data/Part2/TI_horizons_continuous.SGEMS to ti_horizons.dat
```

```
Loading and plotting fluvsim
Beginning download of https://github.com/GAIA-UNIL/trainingimages/raw/master/MPS_book_
↪data/Part2/ti_fluvsim_big_channels3D.zip to ti_fluvsim.dat.zip
Unziping ti_fluvsim.dat.zip to ti_fluvsim.dat
```

Coarsen a 2D training to a 3D

[5]:
```
# load the full Strebelle TI
TI, TI_name = mps.trainingimages.strebelle(di=1)
mps.plot.plot(np.squeeze(TI))

# load every 4th pixel from Strebelle
TI, TI_name = mps.trainingimages.strebelle(di=4)
mps.plot.plot(np.squeeze(TI))
```





[6]:
```
# load every set of possible variation of Strebelle usong a subrgid of 4x4
TI, TI_name = mps.trainingimages.strebelle(di=4,coarse3d=1)
mps.plot.plot(np.squeeze(TI))
```

[ ]:
```

```

9.7.5 MPSlib: estimation

Johansson and Hansen (2021) demonstrate how to perform MPS estimation to directly obtain conditional statistics wiythout th eneed for simulation.

```
O.par['do_estimation'] # [0]: Simulation , [1] estimation
```

See details about MPS estimation in Jóhannsson, Óli D., and Thomas Mejer Hansen. "Estimation using multiple-point statistics." Computers & Geosciences 156 (2021).

[1]:
```
# import mpslib as mps
import matplotlib.pyplot as plt
import numpy as np
import mpslib as mps
```

[2]:
```
O=mps.mpslib(method='mps_genesim',n_max_cpdf_count= 100, simulation_grid_size=np.
↪array([28, 43, 1]));
#O=mps.mpslib(method='mps_snesim_tree', n_multiple_grids=1, simulation_grid_size=np.
↪array([28, 43, 1]));
O.par['verbose_level']=1,


## Set hard data
d_hard = np.array([[ 3, 3, 0, 1],
                   [ 8, 8, 0, 0],
                   [ 12, 3, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
## Set soft data
d_soft = np.array([[ 20, 6, 0, 0.3, 0.7],
                   [ 20, 20, 0, 0.001, 0.999]
                  ])


O.d_hard = d_hard
O.d_soft= d_soft

# Only co-locational
O.par['n_cond_soft']=2


# Set training image
O.ti = mps.trainingimages.strebelle(di=3, coarse3d=1)[0]
```

```
Using mps_genesim installed in /mnt/d/PROGRAMMING/mpslib/scikit-mps/mpslib/bin (scikit-
↪mps in /mnt/d/PROGRAMMING/mpslib/scikit-mps/mpslib/mpslib.py)
```

Estimation

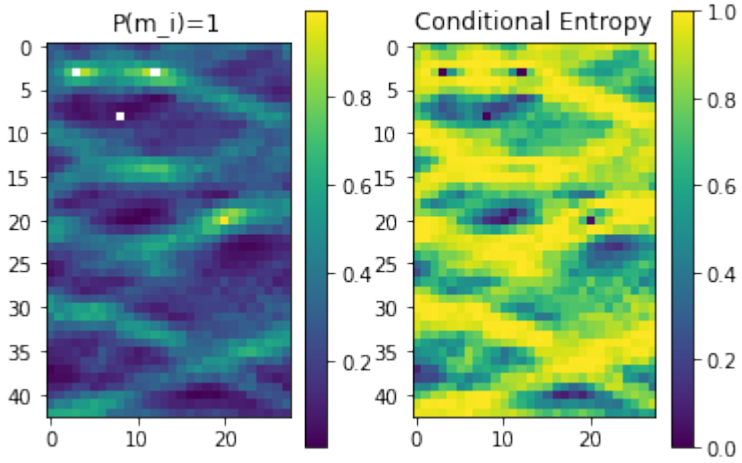
[3]:
```
O.par['n_cond']=3; # For estimation, the numbner of conditionals need never be highed␣
↪than the number of actual conditional hard and/or soft data.
O.par['n_real']=1;
O.par['do_estimation']=1
O.par['do_entropy']=1
# when using mps_genesim, we need to compute a conditional
O.par['n_max_cpdf_count']=1000000; # We need to be able to compute the conditional
O.par['n_max_ite']=1000000
O.delete_local_files() # to make sure no old data are floating around
O.remove_gslib_after_simulation=1

O.run()
print('Time used to perform MPS estimation: %4.1fs' % (O.time))

# Get P(m_i==1)
P1=O.est[1][:,:,0].T
# Get H(m_i
H=O.Hcond[:,:,0].transpose()

plt.figure()
plt.subplot(121)
plt.imshow(P1)
plt.colorbar()
plt.title('P(m_i)=1')
plt.subplot(122)
plt.imshow(H)
plt.colorbar()
plt.title('Conditional Entropy')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

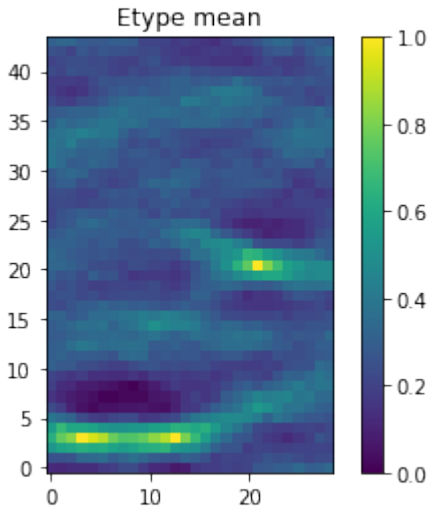
```
loading entropy from ti.dat_ent_0.gslib
loading entropy from ti.dat_cg_0.gslib
loading entropy from ti.dat_cg_1.gslib
Time used to perform MPS estimation: 12.9s
```

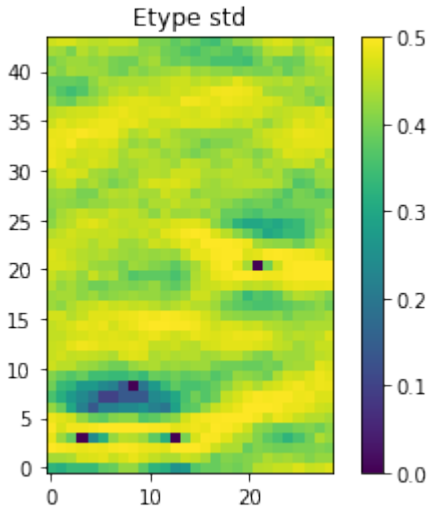


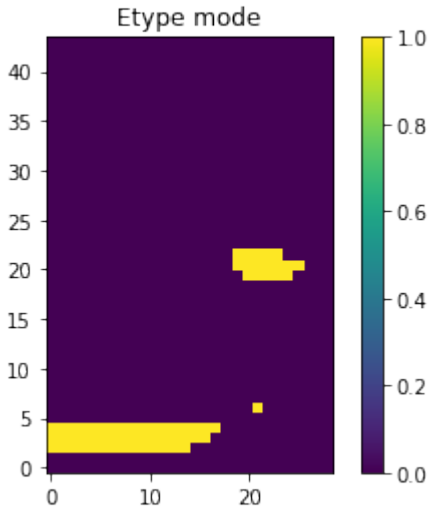
Simulation

[4]:
```
O.par['n_real']=30;
O.par['n_cond']=25; # For estimation, one typically needs n_cond higher to obtain␣
↪realizations with resonable pattern reproduction.

O.par['do_estimation']=0
O.par['max_cpdf_count']=1; # Direct sampling mode
O.par['n_max_ite']=1000 # This is typically not set to a very high number to avoid␣
↪scanning the whole TI

O.par['n_real']=100
O.delete_local_files() # to make sure no old data are floating around
O.remove_gslib_after_simulation=1

O.run_parallel()
print('Time used to perform MPS simulation:%4.1fs' % (O.time))
O.plot_etype()
```

```
parallel: Using 50 of max 52 threads
Time used to perform MPS simulation: 7.2s
```


Etype mean
0
5
10
15
20
25
30
35
40
0
10
20
1.0
0.8
0.6
0.4
0.2
0.0


Etype std
0
5
10
15
20
25
30
35
40
0
10
20
0.5
0.4
0.3
0.2
0.1
0.0


Etype mode
0
5
10
15
20
25
30
35
40
0
10
20
1.0
0.8
0.6
0.4
0.2
0.0

[ ]:

9.7.6 MPSlib: computation of entropy and self-information

The self-information, and entropy (the average self-information), acan be commputed using MPSlib by setting

```
do_entropy=1
```

This works for all algorithms excpet when using mps_genesim in 'direct sampling mode', when O.par['n_max_cpdf_count']=1

See details in

Hansen, Thomas Mejer. "Entropy and information content of geostatistical models." Mathematical Geosciences 53.1 (2021): 163-184

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import mpslib as mps
```

Setup MPSLib

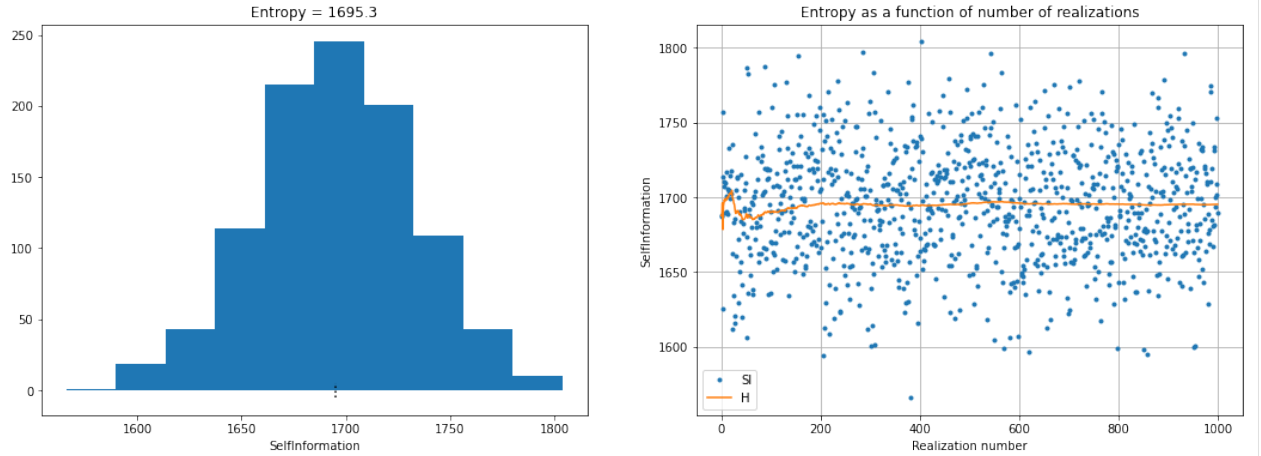
Setup MPSlib, and select to compute entropy using for example

[2]:
```
# Initialize MPSlib using the mps_snesim_tree algorthm, and a simulation grid of size␣
↪[80,70,1]
#O = mps.mpslib(method='mps_genesim', simulation_grid_size=[80,70,1], n_max_cpdf_count=30,
↪ verbose_level=-1)
O = mps.mpslib(method='mps_snesim_tree', simulation_grid_size=[80,70,1], verbose_level=-
↪1)
O.delete_local_files()
O.par['n_real'] = 1000
O.par['n_cond']= 9
# Choose to compute entropy
O.par['do_entropy']=1
TI, TI_filename = mps.trainingimages.strebelle(di=4, coarse3d=1)
O.ti = TI

O_all = O.run_parallel()
```

Plot entropy

[3]:
```
fig = plt.figure(figsize=(18, 6))

plt.subplot(1,2,1)
plt.hist(O.SI)
plt.plot(np.array([1, 1])*O.H,[-5,5],'k:')
plt.xlabel('SelfInformation')
plt.title('Entropy = %3.1f' % (O.H))

plt.subplot(1,2,2)
plt.plot(O.SI,'.', label='SI')

plt.plot(np.cumsum(O.SI)/(np.arange(1,1+len(O.SI))),'-',label='H')
plt.legend()
plt.grid()

plt.xlabel('Realization number')
plt.ylabel('SelfInformation')
plt.title('Entropy as a function of number of realizations' % (O.H))
```

[3]:
```
Text(0.5, 1.0, 'Entropy as a function of number of realizations')
```



Entropy as a function of number of conditional data

[4]:
```
TI, TI_filename = mps.trainingimages.strebelle(di=4, coarse3d=1)

n_cond_arr = np.array([1,2,4,6,8,12,16,24,32,64])

H=np.zeros(n_cond_arr.size) # entropy
t=np.zeros(n_cond_arr.size) # simulation time
i=0
SI=[]
for n_cond in n_cond_arr:
    O = mps.mpslib(method='mps_snesim_tree', simulation_grid_size=[80,70,1], verbose_
↪level=-1)
    O.par['n_real'] = 20
```

(continues on next page)

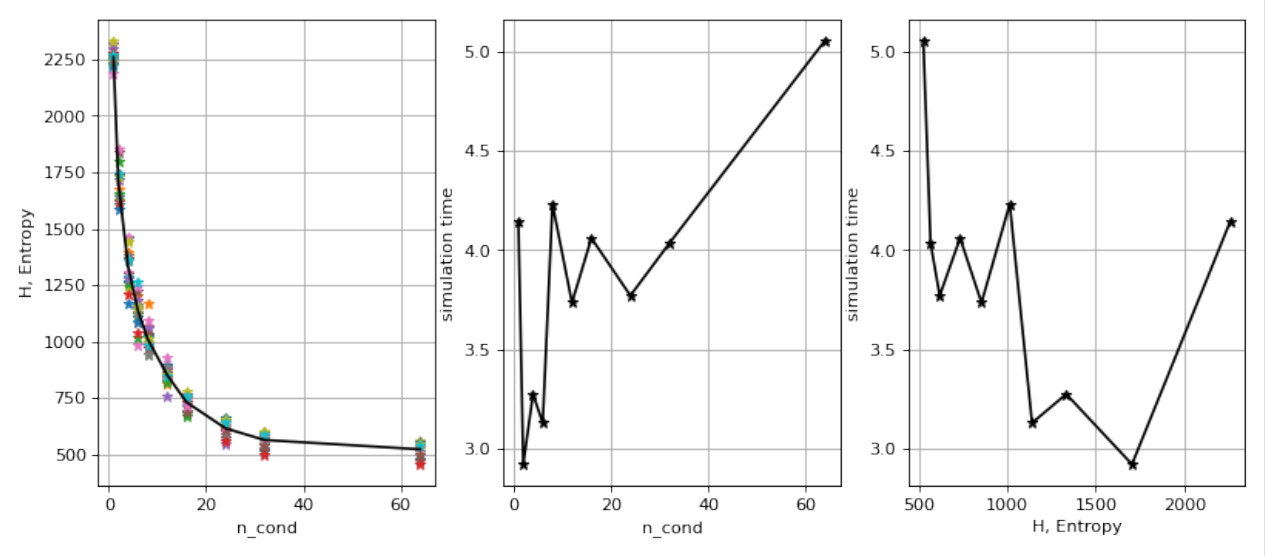
(continued from previous page)

```
    O.par['n_cond']=n_cond
    # Choose to compute entropy
    O.par['do_entropy']=1
    O.TI = TI;

    O.run_parallel()
    #O.run()
    print('n_cond = %d, H=%4.1f' % (n_cond,O.H))
    SI.append(O.SI) # Self-information
    H[i]=O.H # Entropy
    t[i]=O.time
    i=i+1
```

```
n_cond = 1, H=2260.7
n_cond = 2, H=1705.0
n_cond = 4, H=1332.4
n_cond = 6, H=1138.6
n_cond = 8, H=1012.2
n_cond = 12, H=852.4
n_cond = 16, H=730.9
n_cond = 24, H=615.3
n_cond = 32, H=563.6
n_cond = 64, H=522.3
```

[5]:
```
plt.figure(figsize=(12, 5), dpi=80)

ax1 = plt.subplot(1, 3, 1)
plt.plot(n_cond_arr,SI,'*')
plt.plot(n_cond_arr,H,'k-')
plt.grid()
plt.xlabel('n_cond')
plt.ylabel('H, Entropy')

ax2 = plt.subplot(1, 3, 2)
plt.plot(n_cond_arr,t,'k-*')
plt.grid()
plt.xlabel('n_cond')
plt.ylabel('simulation time')

ax3 = plt.subplot(1, 3, 3)
plt.plot(H,t,'k-*')
plt.grid()
plt.xlabel('H, Entropy')
plt.ylabel('simulation time')
```

[5]:
```
Text(0, 0.5, 'simulation time')
```



[ ]:

[ ]:

9.7.7 MPSlib: variable template size in mps_snesim_tree and mps_snesim_list

mps_snesim_tree and mps_snesim_list allowing using a template size that changes for each multiple grid. The template size is given as a value for the coarsest multiple grid, and a value for the final dense simulation grid. Linear interpolation is used to compute the template size at each multiple grid

For example using a template size of 8x7x4 oon the coarsest grid and a template size of 4x3x3 on the finest grid can be given in the mps_snesim parameter file as

```
Search template size X # 8 4
Search template size Y # 7 3
Search template size Z # 4 3
```

This template can be set in Python using

```
O=mps.mpslib(method='mps_snesim_tree')
O.par['template_size']=np.array([[8,7,4],[4,3,3]]).T
```

A simple constant template of size [8,7,3] can be set using

```
O.par['template_size']=np.array([8,7,4])
```

The main reason for using variable template size is that, typically, a considerable amount of CPU is used in the finer simulation grids to prune (remove) conditional data.

The example below demonstrates the CPU time compared to using a varying template size at the finest grid.

See more at https://mpslib.readthedocs.io/en/latest/Examples/ex_varying_template.html

[1]:
```
import mpslib as mps
import numpy as np
```

(continues on next page)

(continued from previous page)

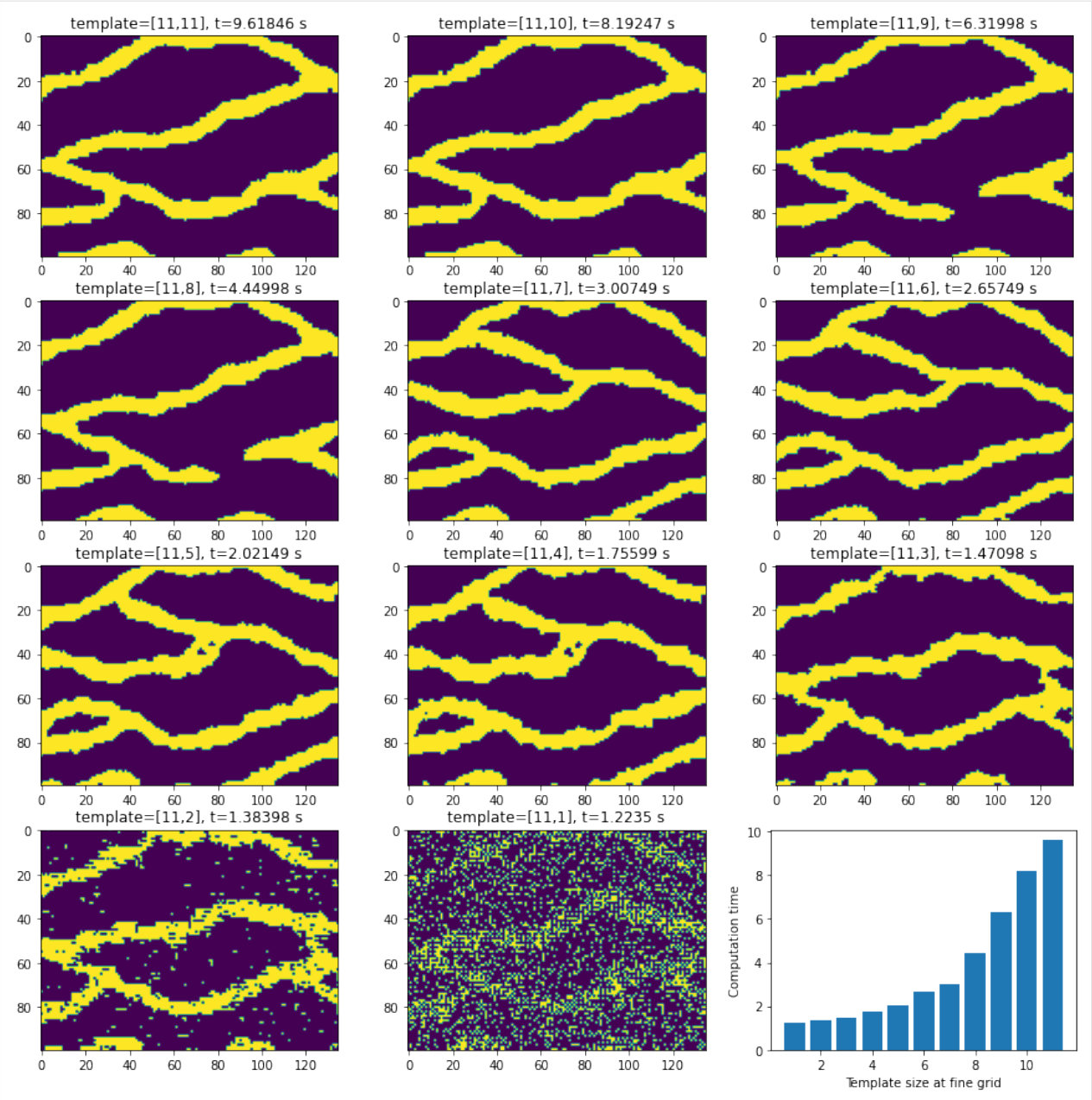
```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
```

[2]:
```
#TI1, TI_filename1 = mps.trainingimages.strebelle(2, coarse3d=1)
TI1, TI_filename1 = mps.trainingimages.strebelle(1, coarse3d=1)

O1=mps.mpslib(method='mps_snesim_tree')
O1.ti=TI1
O1.par['n_multiple_grids']=4;
O1.par['n_cond']=81
O1.par['n_real']=1
O1.par['rseed']=1
O1.par['debug_level']=-1
O1.par['simulation_grid_size'][0]=135
O1.par['simulation_grid_size'][1]=100
O1.par['simulation_grid_size'][2]=1
```

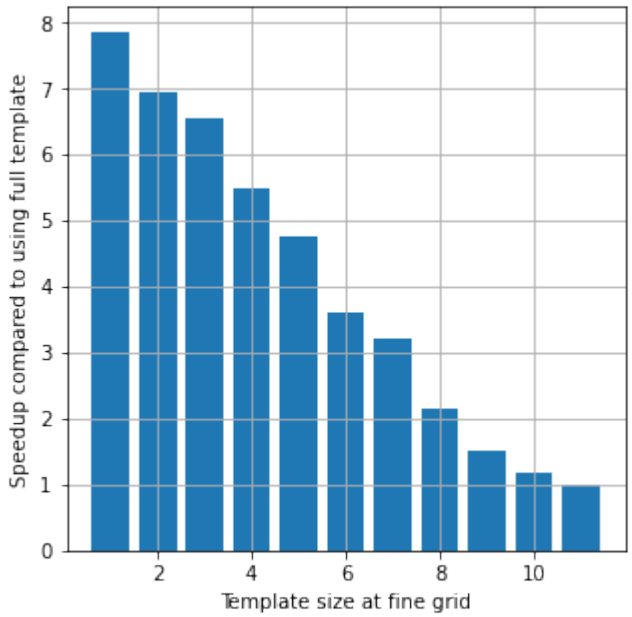
[3]:
```
r1 = 11 # template size in the coarse grid
r2 = [11,10,9,8,7,6,5,4,3,2,1] # template size in the finest grid

t=[]
R=[]
for ir in range(len(r2)):

    O1.delete_local_files()

    template = np.array([[r1, r2[ir]], [r1, r2[ir]], [1, 1]])
    O1.par['template_size']=template
    name = '%s_%d_%d'%(O1.method,r1,r2[ir])
    print(name)
    O1.parameter_filename= name+'.par'
    O1.mps_snesim_par_write()
    O1.run()
    R.append(O1.sim[0])

    t.append(O1.time)
```

```
mps_snesim_tree_11_11
mps_snesim_tree_11_10
mps_snesim_tree_11_9
mps_snesim_tree_11_8
mps_snesim_tree_11_7
mps_snesim_tree_11_6
mps_snesim_tree_11_5
mps_snesim_tree_11_4
mps_snesim_tree_11_3
mps_snesim_tree_11_2
mps_snesim_tree_11_1
```

[4]:
```
#%% Plot the realizations and a bar of the timing
fig = plt.figure(figsize=(15, 15))
outer = gridspec.GridSpec(4, 3, wspace=0.2, hspace=0.2)

for ir in range(len(r2)):
    ax1 = plt.Subplot(fig, outer[ir])
    fig.add_subplot(ax1)
    plt.imshow(np.transpose(np.squeeze(R[ir])))
    plt.title('template=[%d,%d], t=%g s'%(r1,r2[ir],t[ir]))

ax1 = plt.Subplot(fig, outer[-1])
fig.add_subplot(ax1)
plt.bar(r2,t)
plt.xlabel('Template size at fine grid')
plt.ylabel('Computation time')
plt.savefig('varying_template', dpi=600, facecolor='w', edgecolor='w',
        orientation='portrait', transparent=True)

plt.show()
```



[5]:
```
#%% SPPEEDUP
fig = plt.figure(figsize=(5, 5))
plt.bar(r2,t[0]/np.array(t))
plt.xlabel('Template size at fine grid')
plt.ylabel('Speedup compared to using full template')
plt.grid()
plt.savefig('varying_template_speedup', dpi=600, facecolor='w', edgecolor='w',␣
↪orientation='portrait', transparent=True)

plt.show()
```



9.7.8 GENESIM with distance weighing

Mariethoz et al. (2010) propose to weight conditonal data by distance. This is implemented with mps_genesim, and can be controlled by

```
O.par['distance_measure'] # Distance measure [1]: discrete, [2]: continous
O.par['distance_min'] ; # Max distance
O.par['distance_pow'] ; # Power
```

See details about distance weighing in Mariethoz, Gregoire, Philippe Renard, and Julien Straubhaar. "The direct sampling method to perform multiple-point geostatistical simulations." Water Resources Research 46.11 (2010)..

[1]:
```
# import mpslib as mps
import matplotlib.pyplot as plt
import numpy as np
import mpslib as mps
```

[2]:
```
O=mps.mpslib(method='mps_genesim',
             verbose_level=-1,
             n_cond = 25,
             n_real=1,
             simulation_grid_size=np.array([50, 50, 1]));

O.ti, TI_filename = mps.trainingimages.strebelle(4, coarse3d=0)
```

[3]:
```
distance_max_arr = [0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.12, 0.14, 0.16, 0.18, 0.2, 0.22, 0.
↪24, 0.26, 0.28, 0.30]
distance_pow_arr = [0, 1, 2]
```

(continues on next page)

(continued from previous page)

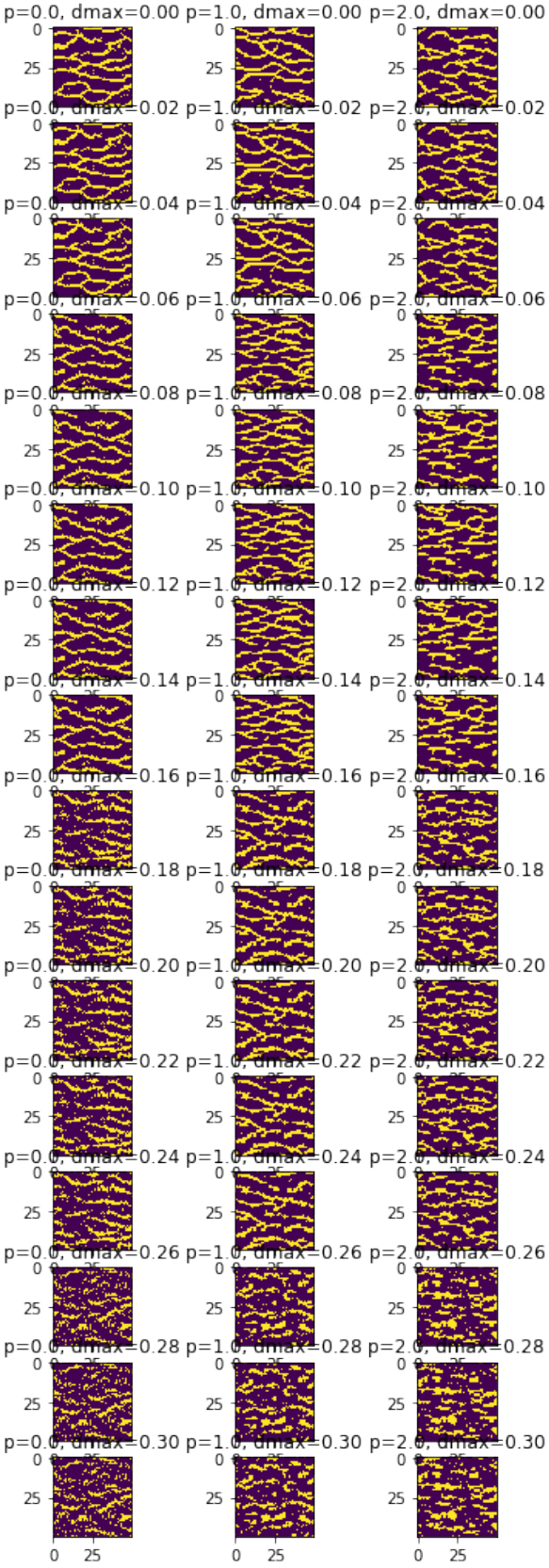
```
n1=len(distance_max_arr)
n2=len(distance_pow_arr)

T=np.zeros((n1,n2))

fig = plt.figure(figsize=(6, 18))
for i1 in np.arange(n1):
    for i2 in np.arange(n2):

        O.par['distance_max']=distance_max_arr[i1]
        O.par['distance_pow']=distance_pow_arr[i2]
        O.par['distance_measure']=1 # discrete
        O.run()
        T[i1,i2]=O.time

        print('distance_max=%g distance_pow=%g, t=%4.2fs' % (O.par['distance_max'],O.par[
↪'distance_pow'],T[i1,i2]))


        isp = i1*n2+i2+1
        plt.subplot(n1,n2,isp)
        D=np.squeeze(np.transpose(O.sim[0]));
        plt.imshow(D, interpolation='none', vmin=0, vmax=1)
        plt.title('p=%3.1f, dmax=%3.2f' % (O.par['distance_pow'],O.par['distance_max']),␣
↪)
```

```
distance_max=0 distance_pow=0, t=1.97s
distance_max=0 distance_pow=1, t=5.51s
distance_max=0 distance_pow=2, t=5.72s
distance_max=0.02 distance_pow=0, t=1.96s
distance_max=0.02 distance_pow=1, t=5.54s
distance_max=0.02 distance_pow=2, t=5.62s
distance_max=0.04 distance_pow=0, t=1.96s
distance_max=0.04 distance_pow=1, t=5.60s
distance_max=0.04 distance_pow=2, t=5.67s
distance_max=0.06 distance_pow=0, t=0.72s
distance_max=0.06 distance_pow=1, t=1.56s
distance_max=0.06 distance_pow=2, t=0.68s
distance_max=0.08 distance_pow=0, t=0.64s
distance_max=0.08 distance_pow=1, t=1.56s
distance_max=0.08 distance_pow=2, t=0.69s
distance_max=0.1 distance_pow=0, t=0.65s
distance_max=0.1 distance_pow=1, t=1.56s
distance_max=0.1 distance_pow=2, t=0.68s
distance_max=0.12 distance_pow=0, t=0.68s
distance_max=0.12 distance_pow=1, t=1.57s
distance_max=0.12 distance_pow=2, t=0.67s
distance_max=0.14 distance_pow=0, t=0.63s
distance_max=0.14 distance_pow=1, t=1.55s
distance_max=0.14 distance_pow=2, t=0.65s
distance_max=0.16 distance_pow=0, t=0.14s
distance_max=0.16 distance_pow=1, t=0.20s
```

(continues on next page)

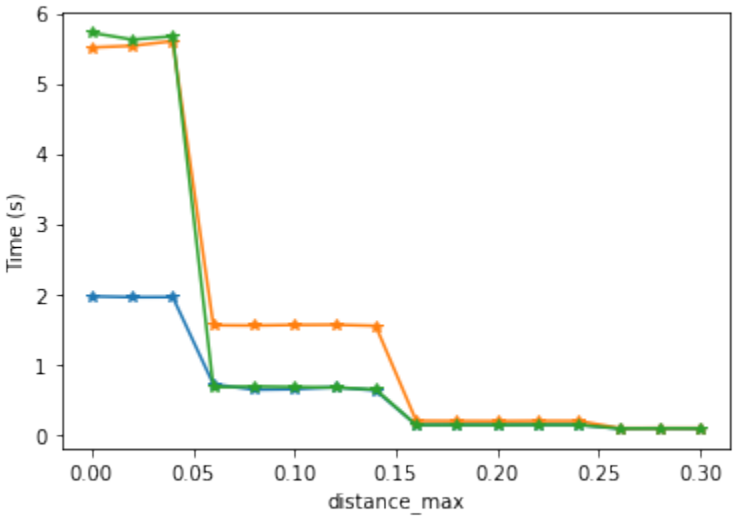
(continued from previous page)

```
distance_max=0.16 distance_pow=2, t=0.14s
distance_max=0.18 distance_pow=0, t=0.14s
distance_max=0.18 distance_pow=1, t=0.20s
distance_max=0.18 distance_pow=2, t=0.14s
distance_max=0.2 distance_pow=0, t=0.14s
distance_max=0.2 distance_pow=1, t=0.20s
distance_max=0.2 distance_pow=2, t=0.14s
distance_max=0.22 distance_pow=0, t=0.14s
distance_max=0.22 distance_pow=1, t=0.20s
distance_max=0.22 distance_pow=2, t=0.14s
distance_max=0.24 distance_pow=0, t=0.14s
distance_max=0.24 distance_pow=1, t=0.20s
distance_max=0.24 distance_pow=2, t=0.14s
distance_max=0.26 distance_pow=0, t=0.09s
distance_max=0.26 distance_pow=1, t=0.10s
distance_max=0.26 distance_pow=2, t=0.09s
distance_max=0.28 distance_pow=0, t=0.09s
distance_max=0.28 distance_pow=1, t=0.10s
distance_max=0.28 distance_pow=2, t=0.09s
distance_max=0.3 distance_pow=0, t=0.09s
distance_max=0.3 distance_pow=1, t=0.10s
distance_max=0.3 distance_pow=2, t=0.09s
```


p=0.0, dmax=0.00
p=1.0, dmax=0.00
p=2.0, dmax=0.00
p=0.0, dmax=0.02
p=1.0, dmax=0.02
p=2.0, dmax=0.02
p=0.0, dmax=0.04
p=1.0, dmax=0.04
p=2.0, dmax=0.04
p=0.0, dmax=0.06
p=1.0, dmax=0.06
p=2.0, dmax=0.06
p=0.0, dmax=0.08
p=1.0, dmax=0.08
p=2.0, dmax=0.08
p=0.0, dmax=0.10
p=1.0, dmax=0.10
p=2.0, dmax=0.10
p=0.0, dmax=0.12
p=1.0, dmax=0.12
p=2.0, dmax=0.12
p=0.0, dmax=0.14
p=1.0, dmax=0.14
p=2.0, dmax=0.14
p=0.0, dmax=0.16
p=1.0, dmax=0.16
p=2.0, dmax=0.16
p=0.0, dmax=0.18
p=1.0, dmax=0.18
p=2.0, dmax=0.18
p=0.0, dmax=0.20
p=1.0, dmax=0.20
p=2.0, dmax=0.20
p=0.0, dmax=0.22
p=1.0, dmax=0.22
p=2.0, dmax=0.22
p=0.0, dmax=0.24
p=1.0, dmax=0.24
p=2.0, dmax=0.24
p=0.0, dmax=0.26
p=1.0, dmax=0.26
p=2.0, dmax=0.26
p=0.0, dmax=0.28
p=1.0, dmax=0.28
p=2.0, dmax=0.28
p=0.0, dmax=0.30
p=1.0, dmax=0.30
p=2.0, dmax=0.30
0
25

[4]:
```
plt.plot(distance_max_arr,T,'-*')
plt.xlabel('distance_max')
plt.ylabel('Time (s)')
#plt.legend()
```

[4]:
```
Text(0, 0.5, 'Time (s)')
```



[5]:
```
O.par
```

[5]:
```
{'n_real': 1,
 'rseed': 1,
 'n_max_cpdf_count': 1,
 'out_folder': '.',
 'ti_fnam': 'ti.dat',
 'simulation_grid_size': array([50, 50,  1]),
 'origin': array([0., 0., 0.]),
 'grid_cell_size': array([1, 1, 1]),
 'mask_fnam': 'mask.dat',
 'hard_data_fnam': 'hard.dat',
 'shuffle_simulation_grid': 2,
 'entropyfactor_simulation_grid': 4,
 'shuffle_ti_grid': 1,
 'hard_data_search_radius': 1,
 'soft_data_categories': array([0, 1]),
 'soft_data_fnam': 'soft.dat',
 'n_threads': -1,
 'debug_level': -1,
 'do_estimation': 0,
 'do_entropy': 0,
 'n_cond': 25,
 'n_cond_soft': 1,
 'n_max_ite': 1000000,
 'distance_measure': 1,
 'distance_max': 0.3,
 'distance_pow': 2,
```

(continues on next page)

(continued from previous page)

```
 'colocate_dimension': 0,
 'max_search_radius': 10000000,
 'max_search_radius_soft': 10000000}
```

[ ]:

9.7.9 Example: Mapping buried valleys in Kasted, Denmark

[1]:
```
%load_ext autoreload
%autoreload 2
import mpslib as mps
import numpy as np
import matplotlib.pyplot as plt
```

Get the training image and conditional data

[2]:
```
# Get the training image
dx=400 # Lowest resolution - faster
dx=200
#dx=100
#dx=50 # highest resolution - slower
TI, TI_fname = mps.trainingimages.kasted(dx=dx)

url_base = 'https://raw.githubusercontent.com/ergosimulation/mpslib/master/data/kasted'
remote_files=['kasted_soft_well.dat',  'kasted_soft_ele.dat',  'kasted_soft_res.dat',
↪'kasted_hard_well_consistent.dat' ]
for local_file in remote_files:
    mps.trainingimages.get_remote('%s/%s' %(url_base,local_file), local_file)

# Get conditional point data
EAS_well=mps.eas.read(remote_files[0])
EAS_ele=mps.eas.read(remote_files[1])
EAS_res=mps.eas.read(remote_files[2])
EAS_well_hard=mps.eas.read(remote_files[3])
```

[3]:
```
# Setup the geometry
x_pad = 4*dx
x_min = np.min(EAS_well['D'][:,0])-x_pad
x_max = np.max(EAS_well['D'][:,0])+x_pad
y_min = np.min(EAS_well['D'][:,1])-x_pad
y_max = np.max(EAS_well['D'][:,1])+x_pad
z_min = np.min(EAS_well['D'][:,2])
z_max = np.max(EAS_well['D'][:,2])
x_min = dx*np.floor(x_min/dx)
y_min = dx*np.floor(y_min/dx)
z_min = dx*np.floor(z_min/dx)
```

(continues on next page)

(continued from previous page)

```
nx=np.int16(np.ceil((x_max-x_min)/dx))
ny=np.int16(np.ceil((y_max-y_min)/dx))
nz=np.max([np.int16(np.ceil((z_max-z_min)/dx)),1])

grid_cell_size = np.array([1, 1, 1])*dx
origin = np.array([x_min, y_min, z_min])

simulation_grid_size=np.array([nx, ny, nz])

print('Origin = [x0,y0,z0]=[%g,%g,%g]' % (origin[0],origin[1],origin[2]))
print('Grid cell size [dx,dy,dz]=[%g,%g,%g]' % (grid_cell_size[0],grid_cell_size[1],grid_
↪cell_size[2]))
print('Simulation Grid size [nx,ny,nz]=[%d,%d,%d]' % (simulation_grid_size[0],
↪simulation_grid_size[1],simulation_grid_size[2]))

print('X:[%g,%g] Y:[%g,%g] Z:[%g,%g]' % (x_min, x_max, y_min, y_max, z_min, z_max))
```

```
Origin = [x0,y0,z0]=[560800,6.2246e+06,0]
Grid cell size [dx,dy,dz]=[200,200,200]
Simulation Grid size [nx,ny,nz]=[82,58,1]
X:[560800,577187] Y:[6.2246e+06,6.2362e+06] Z:[0,0]
```

[ ]:

Plot the training image and the conditional data

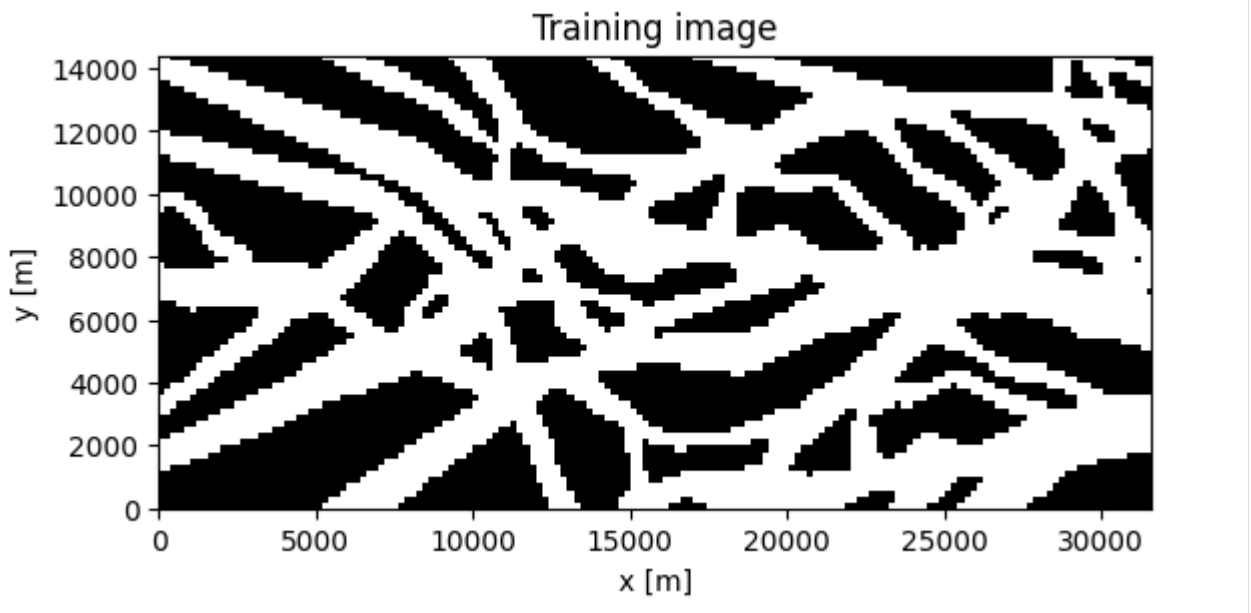
[4]:
```
# Create a figure with 2x3 subplot

# TI
plt.figure()
plt.imshow(TI[:,:,0].T, cmap='gray', origin='lower', extent=[0, dx*TI.shape[0], 0, dx*TI.
↪shape[1]])
plt.title('Training image')
plt.xlabel('x [m]')
plt.ylabel('y [m]')

# Conditional data
fig, axs = plt.subplots(2, 2, figsize=(20, 10))

# Soft well data
scatter_soft = axs[0, 0].scatter(EAS_well['D'][:,0], EAS_well['D'][:,1], c=EAS_well['D
↪'][:,3], s=20, vmin=0, vmax=1)
axs[0, 0].set_title('Soft well data - %s' % EAS_well['header'][3])
axs[0, 0].set_xlabel(EAS_well['header'][0])
axs[0, 0].set_xlabel(EAS_well['header'][1])
axs[0, 0].set_aspect('equal', 'box')
fig.colorbar(scatter_soft, ax=axs[0, 0])

# Hard well data
scatter_hard = axs[0, 1].scatter(EAS_well_hard['D'][:,0], EAS_well_hard['D'][:,1], c=EAS_
↪well_hard['D'][:,3], s=20)
```

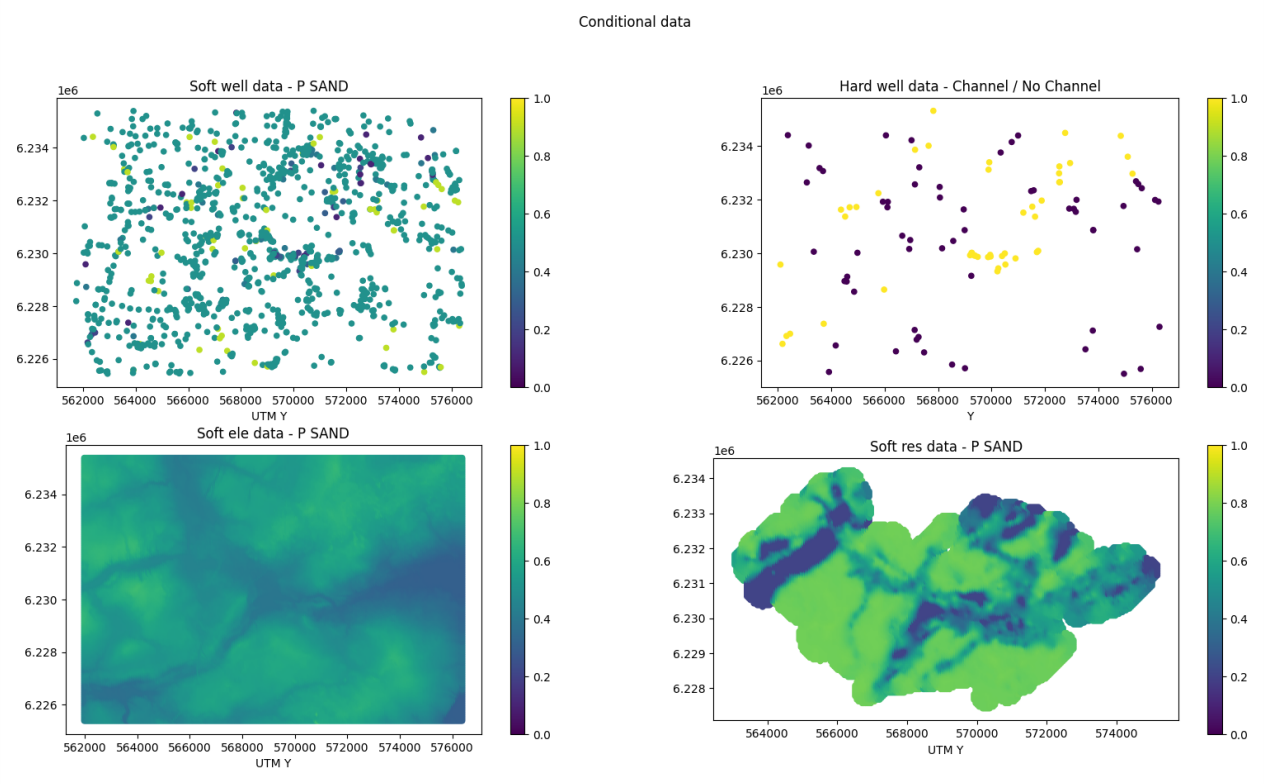
(continues on next page)

(continued from previous page)

```
axs[00, 1].set_title('Hard well data - %s' % EAS_well_hard['header'][3])
axs[0, 1].set_xlabel(EAS_well_hard['header'][0])
axs[0, 1].set_xlabel(EAS_well_hard['header'][1])
axs[0, 1].set_aspect('equal', 'box')
fig.colorbar(scatter_hard, ax=axs[0, 1])

# ELE data
scatter_ele = axs[1, 0].scatter(EAS_ele['D'][:,0], EAS_ele['D'][:,1], c=EAS_ele['D'][:,
↪3], s=20, vmin=0, vmax=1)
axs[1, 0].set_title('Soft ele data - %s' % EAS_ele['header'][3])
axs[1, 0].set_xlabel(EAS_ele['header'][0])
axs[1, 0].set_xlabel(EAS_ele['header'][1])
axs[1, 0].set_aspect('equal', 'box')
fig.colorbar(scatter_ele, ax=axs[1, 0])

# RES data
scatter = axs[1, 1].scatter(EAS_res['D'][:,0], EAS_res['D'][:,1], c=EAS_res['D'][:,3],␣
↪s=20, vmin=0, vmax=1)
axs[1, 1].set_title('Soft res data - %s' % EAS_res['header'][3])
axs[1, 1].set_xlabel(EAS_res['header'][0])
axs[1, 1].set_xlabel(EAS_res['header'][1])
axs[1, 1].set_aspect('equal', 'box')
fig.colorbar(scatter, ax=axs[1, 1])

fig.suptitle('Conditional data')
```

[4]:
```
Text(0.5, 0.98, 'Conditional data')
```





Optionally remove non-informed conditional well data

[5]:

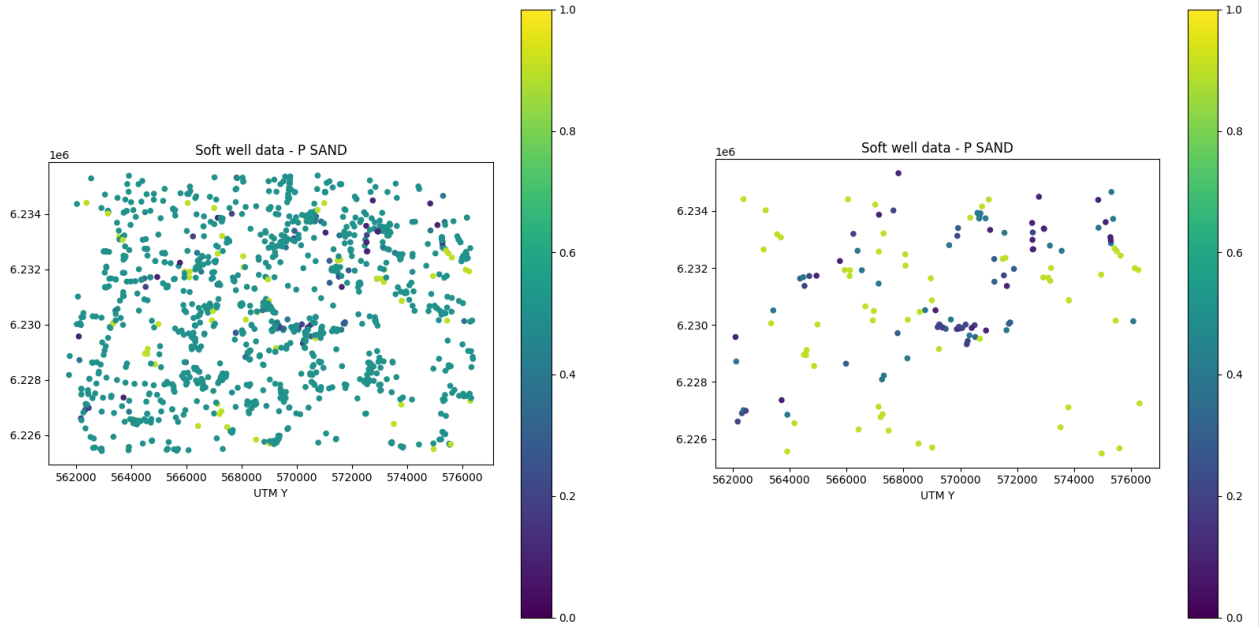
```
# Optionally remove non-informed conditional well data
RemoveUninformed = True

if RemoveUninformed:

    fig, axs = plt.subplots(1, 2, figsize=(20, 10))
    # Before
    scatter_soft = axs[0].scatter(EAS_well['D'][:,0], EAS_well['D'][:,1], c=EAS_well['D
↪'][:,3], s=20, vmin=0, vmax=1)
    axs[0].set_title('Soft well data - %s' % EAS_well['header'][3])
    axs[0].set_xlabel(EAS_well['header'][0])
    axs[0].set_xlabel(EAS_well['header'][1])
    axs[0].set_aspect('equal', 'box')
    fig.colorbar(scatter_soft, ax=axs[0])


    print(EAS_well['D'].shape)
    informed_well_soft_data=np.argwhere(np.abs(EAS_well['D'][:,4]-0.5)>0.05)
    i_use=informed_well_soft_data.flatten()
    EAS_well['D']=EAS_well['D'][i_use,:]
    print(EAS_well['D'].shape)

    # After
```

(continues on next page)

(continued from previous page)

```
    scatter_soft = axs[1].scatter(EAS_well['D'][:,0], EAS_well['D'][:,1], c=EAS_well['D
↪'][:,3], s=20, vmin=0, vmax=1)
    axs[1].set_title('Soft well data - %s' % EAS_well['header'][3])
    axs[1].set_xlabel(EAS_well['header'][0])
    axs[1].set_xlabel(EAS_well['header'][1])
    axs[1].set_aspect('equal', 'box')
    fig.colorbar(scatter_soft, ax=axs[1])
```

```
(1254, 5)
(147, 5)
```



9.7.10 MPSlib in Kasted

Given the data above, the challenge is to estimate the probability that a buried valley exists in the area.

Setup and run MPSlib

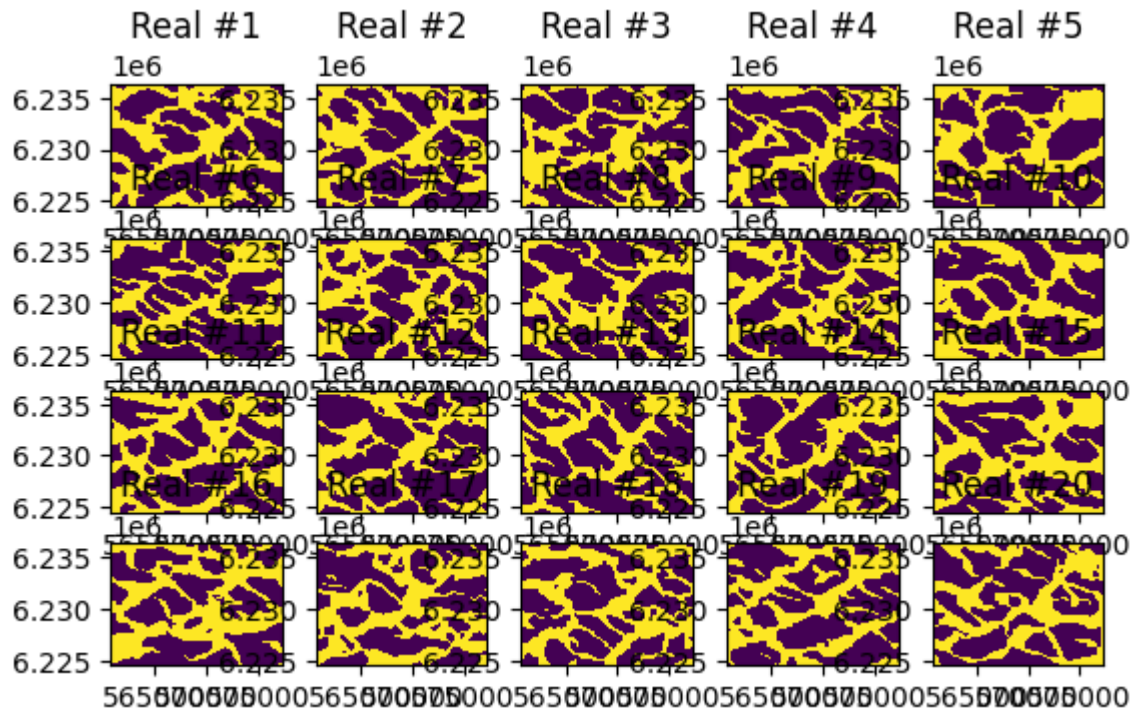
[6]:
```
method = 'mps_genesim'
method = 'mps_snesim_tree'
O=mps.mpslib(method=method,
             simulation_grid_size=simulation_grid_size,
             origin=origin,
             grid_cell_size=grid_cell_size)
O.par['n_real']=20
O.ti=TI

O.run_parallel()
```

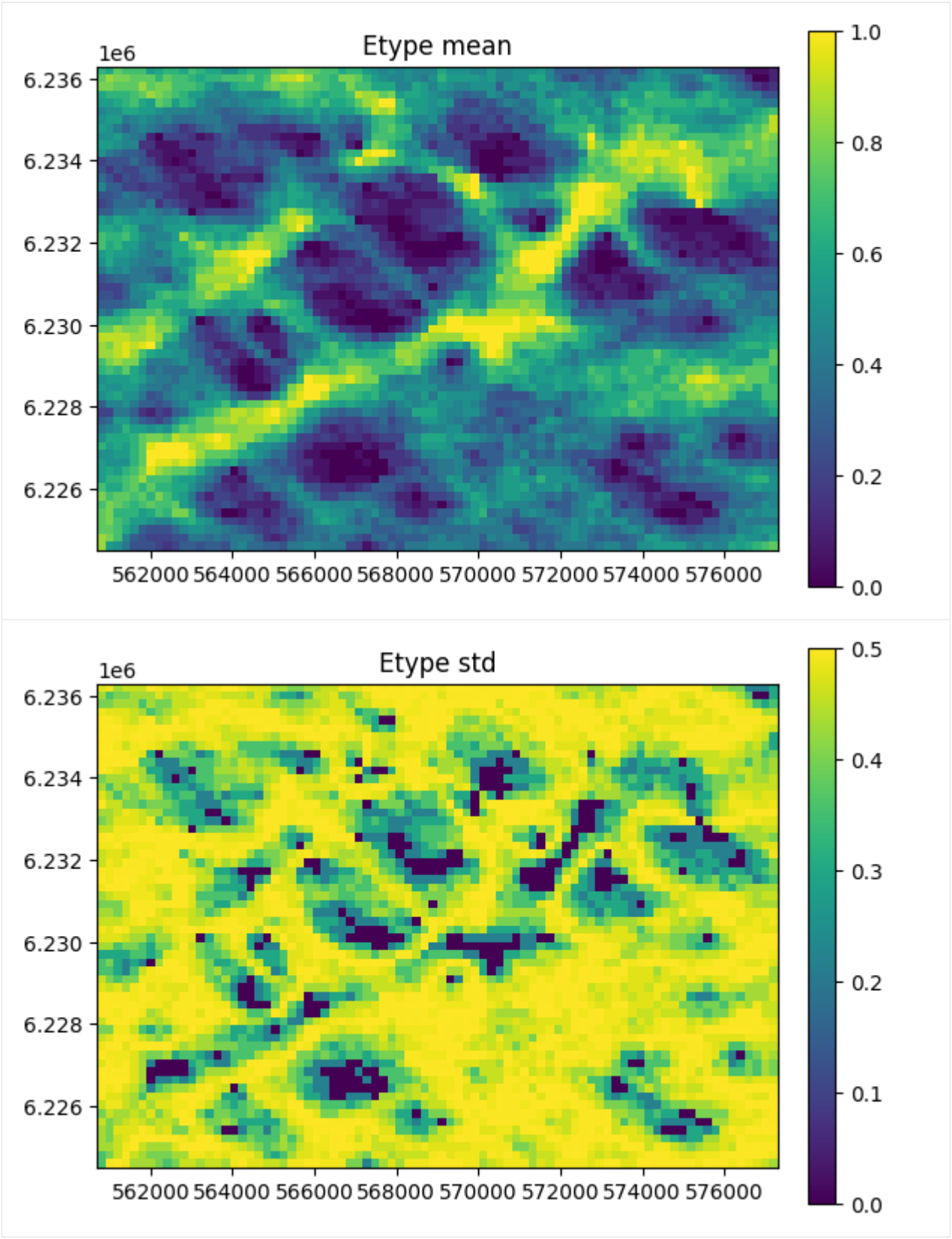
(continues on next page)

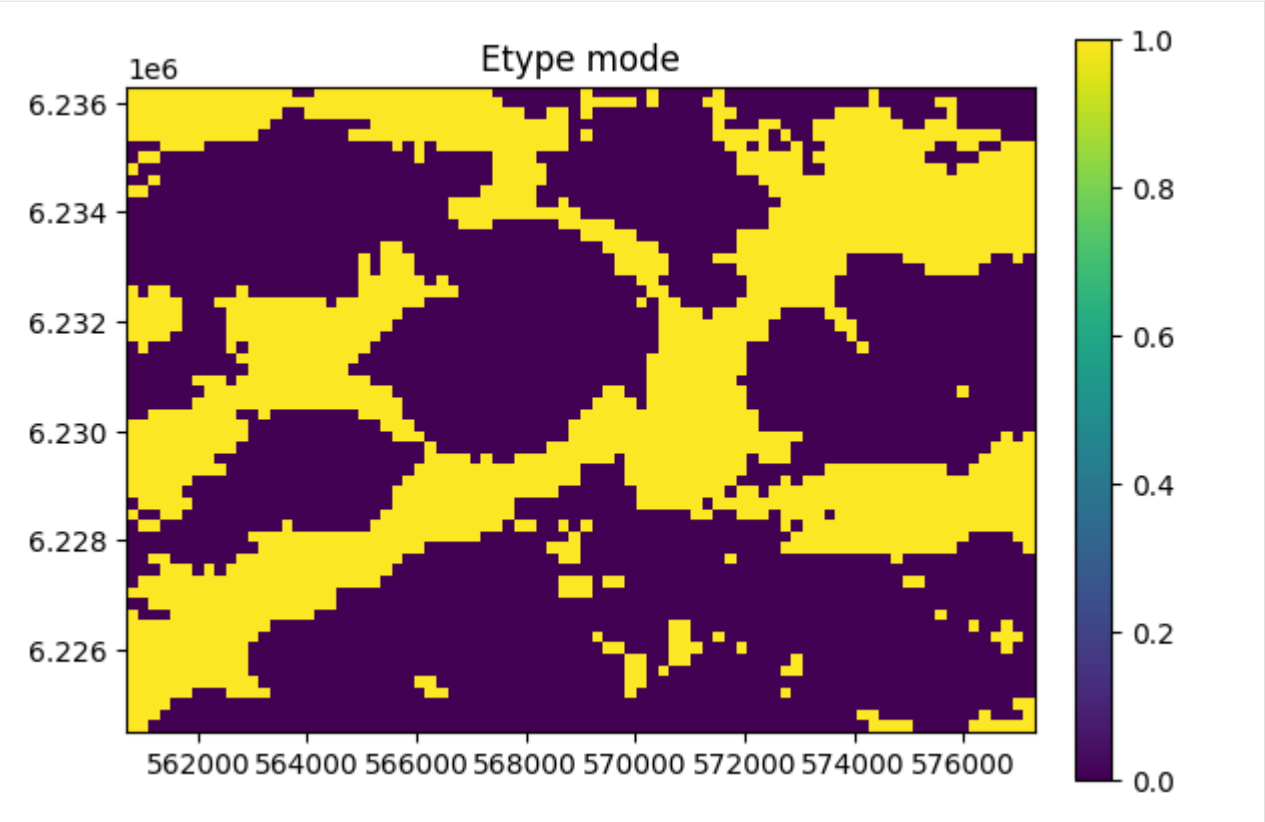
(continued from previous page)

```
O.plot_reals()

O.plot_etype()
```

```
Using mps_snesim_tree installed in /mnt/space/space_au11687/PROGRAMMING/mpslib (scikit-
↪mps in /mnt/space/space_au11687/PROGRAMMING/mpslib/scikit-mps/mpslib/mpslib.py)
parallel: Using 20 of max 26 threads
```




Etype mean
1e6
6.236
6.234
6.232
6.230
6.228
6.226
562000 564000 566000 568000 570000 572000 574000 576000
1.0
0.8
0.6
0.4
0.2
0.0
Etype std
1e6
6.236
6.234
6.232
6.230
6.228
6.226
562000 564000 566000 568000 570000 572000 574000 576000
0.5
0.4
0.3
0.2
0.1
0.0



9.7.11 Conditional simulation - hard data

Adjust the simulation parameters to use hard conditional data. For help take a look at the documentation, and https://github.com/ergosimulation/mpslib/blob/master/scikit-mps/examples/ex_mpslib_hard_and_soft.ipynb

[ ]:
```
method = 'mps_genesim'
method = 'mps_snesim_tree'
O=mps.mpslib(method=method,
             simulation_grid_size=simulation_grid_size,
             origin=origin,
             grid_cell_size=grid_cell_size)
O.ti=TI

O.d_hard = EAS_well_hard['D']

O.plot_hard()
```

[ ]:
```
O.par['n_real']=4
O.ti=TI

O.run_parallel()

O.plot_reals()

O.plot_etype()
```

9.7.12 Conditional simulation - soft data

Adjust the simulation parameters to use soft conditional data

[ ]:
```
method = 'mps_genesim'
method = 'mps_snesim_tree'
O=mps.mpslib(method=method,
             simulation_grid_size=simulation_grid_size,
             origin=origin,
             grid_cell_size=grid_cell_size)
O.ti=TI

# O.d_hard = EAS_well_hard['D']

O.d_soft = EAS_well['D']

O.plot_soft()
```

9.7.13 Conditional simulation - Setup MPSlib to use both conditional hard well data, aoft conditional data related to ELEVATION and RESISTIVITY

[ ]:

[ ]:

9.7.14 Conditional estimation

Setup MPS lib to perform sequential ESTIMATION, of the cases considered above. Get ideas from https://github.com/ergosimulation/mpslib/blob/master/scikit-mps/examples/ex_mpslib_estimation.ipynb

[ ]:

9.8 Implementation

MPSlib is developed in C++, that provides both a C++ class, with wich different sequential simulation algorithms can be implemented, and three example implementations.

Details about the design of the mpslib C++ class can be found in [HANSEN2016]. (link).

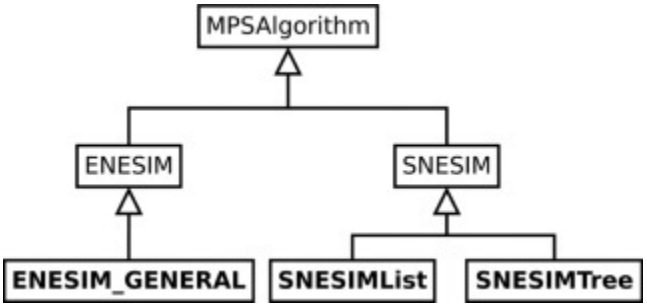
`MPS` is a namespace that contains different classes.

Briefly described, the main class is the `MPSalgorithm` class, which implements the sequential simulation algorithm (using multiple girds), methods for reading and writing 3D gridded data, methods for reading known data values (known as hard and soft data), and methods for establishing a data neighborhood, and controlling the simulation path.

Two abstract member functions are defined, but not implemented: `MPSAlgorithm::readConfig` and `MPSAlgorithm::simulate`

In order to implement a spceific algorithm one must therefore needs to create a class that inherits MPSAlgorithm, and that implements MPSAlgorithm::readConfig and MPSAlgorithm::simulate.

Two subclasses, `ENESIM` and `SNESIM` extends `MPSalgorithm` to allow ENESIM and SNESIM type simulation. Finally, the three core algorithms are implemented based on these classes.



9.8.1 EX: The ENESIM Class

The ENESIM CLASS is inherited from the MPSAlgorithm class, in the MPS namespace

ENESIM_GENERAL.cpp/h Inherits the ENESIM cass from the MPSAlgorithm class, and has two methods:

```
ENESIM_GENERAL::initialize
ENESIM_GENERAL::startSimulation (inherited from MPSalgorithm
```

ENESIM_GENERAL contain implementation of 'readConfiguration' and 'simulate'

ENESIM_GENERAL.h includes mpslib/ENESIM.h which contains the implementation of 'readConfiguration' and 'simulate' MPS::ENESIM_GENERAL::_simulate

mpslib/ENESIM.cpp/h implements the method MPS::ENESIM_GENERAL::_readConfigurations and the function '_getRealizationFromCpdfTiEnesimRejectionNonCo' which is in effect what os evaluated in the MPS::ENESIM_GENERAL::_simulate

mps_genesim.cpp/h simple load the class and call the method 'startSimulation' which is defined in MPSalgorithm

9.9 Contributions

MPSlib was originally co-developed in 2016 by Le Thanh Vu, Torben Bach (I-GIS) and Thomas Mejer Hansen (now Aarhus University)

2017-05-03 Troels Norvin Vilhelmsen (Aarhus University) added a Pyhton interface

2016-10-16: Mathieu Gravey (GAIA Lab, Université de Lausanne) added fixes for clean comilation in OSX

9.10 References

BIBLIOGRAPHY

[GUARDIANO] Guardiano, F. B., & Srivastava, R. M. (1993). Multivariate geostatistics: beyond bivariate moments. In Geostatistics Troia'92 (pp. 133-144). Springer, Dordrecht.

[HANSEN2016] Hansen, T.M., Vu. L.T., and Bach, T. 2016. MPSLIB: A C++ class for sequential simulation of multiple-point statistical models, in *SoftwareX*, doi:10.1016/j.softx.2016.07.001. [pdf,www].

[HANSEN2018] Hansen, T. M., Mosegaard, K., & Cordua, K. S. (2018). Multiple point statistical simulation using uncertain (soft) conditional data. *Computers & geosciences*, 114, 1-10. doi:10.1016/j.cageo.2018.01.017.

[HANSEN2020] Hansen, T. M. Entropy and information content of geostatistical models (2020). Mathematical Geosciences: 1-22. doi:*10.1007/s11004-020-09876-z <https://doi.org/10.1007/s11004-020-09876-z>*.

[JOHANNSSON2021] Jóhannsson, Óli D., O. and Hansen, T. M. (2021). Estimation using multiple-point statistics. Computers & Geosciences 156. doi:*10.1016/j.cageo.2021.104894 <https://doi.org/10.1016/j.cageo.2021.104894>*.

[MARIETHOZ2010] Mariethoz, G., Renard, P., & Straubhaar, J. (2010). The direct sampling method to perform multiple-point geostatistical simulations. Water Resources Research, 46(11).

[STRAUBHAAR2011] Straubhaar, J., Renard, P., Mariethoz, G., Froidevaux, R., & Besson, O. (2011). An improved parallel multiple-point algorithm using a list approach. Mathematical Geosciences, 43(3), 305-328.

[STREBELLE2002] Strebelle, S. (2002). Conditional simulation of complex geological structures using multiple-point statistics. Mathematical geology, 34(1), 1-21.