MORPHOLOGICAL SPATIAL PATTERN ANALYSIS: OPEN SOURCE RELEASE

Pierre Soille 1*, Peter Vogt 1

1 European Commission, Joint Research Centre (JRC), Ispra, Italy - (Pierre.Soille, Peter.Vogt)@ec.europa.eu

Commission IV, WG IV/4

KEY WORDS: spatial pattern analysis, connectivity, mathematical morphology, morphological image analysis, landscape analysis, MSPA, open source, open science.

ABSTRACT:

The morphological segmentation of binary patterns provides an effective method for characterising spatial patterns with emphasis on connections between their parts as measured at varying analysis scales. The method is widely used for the analysis of landscape patterns such as those related to the fragmentation of forests or other natural land cover classes. This can be explained by its effectiveness at capturing the complexity of binary patterns and their connections by partitioning the foreground pixels of the corresponding binary images into mutually exclusive classes. While the principles of the method are conceptually simple, the definition of the classes relies on a series of advanced mathematical morphology operations whose actual implementation is not straightforward. In this paper, we propose an open source code for MSPA and detail its main components in the form of pseudo-code. We demonstrate its effectiveness for asynchronous processing of tera-pixel images and the synchronous exploratory analysis and rendering with Jupyter notebooks.

1. INTRODUCTION

The morphological segmentation of binary patterns (Soille and Vogt, 2009) provides an effective method for characterising spatial patterns with emphasis on connections between their parts as measured at varying analysis scales. The method is now widely used for the analysis of landscape patterns such as those related to the fragmentation of forests or other natural land cover classes, e.g., (Ossola et al., 2019; Carlier et al., 2020; Rincón et al., 2021; Modica et al., 2021). This can be explained by its effectiveness at capturing the complexity of binary patterns and their connections by partitioning the foreground and background pixels of the corresponding binary images into mutually exclusive classes with a clear semantic meaning. While the principles of the method are conceptually simple, the formal definition of the classes relies on a series of advanced mathematical morphology operations whose actual implementation is not straightforward. This issue was originally addressed by the authors by offering a compiled, standalone version named Morphological Spatial Pattern Analysis (MSPA), which is distributed within the applications GuidosToolbox (GTB) (Vogt and Ritters, 2017), GuidosToolbox Workbench (GWB) (Vogt et al., 2022), and various GIS-extensions, see the MSPA home page1 for additional details and application examples.

In this paper, we propose an open source code for MSPA and describe its components for the extraction of all pixel classes (Sec. 2). We then demonstrate its effectiveness for asynchronous processing of tera-pixel images and the synchronous exploratory analysis and rendering with Jupyter notebooks (Sec. 3). Concluding remarks are presented in Sec. 4.

2. MSPA CODE DESCRIPTION

In this section, we describe the main routines of the MSPA code with reference to the morphological image analysis operations they rely on with links to their implementation in the open source Morphological Image Analysis Library (miallib) recently released on GitHub2. Note that numerous functions of miallib support multi-threading based on OpenMP3. Unless a specific reference is provided, all morphological image analysis operators that are referred to hereafter are described in (Soille, 2004).

A synthetic input binary image with foreground (grey shaded) and background (white) pixels together with its corresponding 7 foreground and 3 background MSPA classes is displayed in Figure 1. The 7 MSPA foreground classes and 3 background classes with reference to the source code of the main morphological image analysis function used to compute them are presented hereafter. The underlying MSPA code in the C programming language is available at GitHub4.

2.1 Input and output

The miallib function implementing the morphological segmentation of binary patterns (Soille and Vogt, 2009) is named segmentBinaryPatterns:

```
IMAGE *segmentBinaryPatterns (IMAGE *imin, float size, int graphfg, int transition, int internal)
```

The function segmentBinaryPatterns requires five input parameters defined as follows:

1. imin: an input raster image with pixels of type unsigned char and with foreground pixels set to 2, background pixels set to 1, and no data pixels set to 0;

---

* Corresponding author
2. size: a float number greater or equal to 1 indicating the width of the edges;

3. graphfg: an integer with value 4 or 8 indicating the connectivity rule between adjacent foreground pixels. Note that the connectivity rule for background pixels (defined as graphbg in mspa.c) is dual to that considered for foreground pixels. That is, if graphfg is set to 8, then graphbg will be set to 4 and vice versa;

4. transition: a Boolean value indicating how transition pixels should be displayed;

5. internal: a Boolean value indicating how embedded components should be processed (0 for no special treatment, 1 for assigning special values to pixels belonging to embedded components (like core components fully surrounded by a larger core component).

The function segmentBinaryPatterns returns an output raster image with pixels of type unsigned char and pixel values matching the class of the foreground and background pixels of the input image and given the values of the additional 4 input parameters.

The input and output raster images of the function segmentBinaryPatterns is of type IMAGE, i.e., a miallib raster image. It is defined as C structure containing basic image raster information such as dimensions as well as pointer to a one-dimensional array holding the pixel values of the image. For binding the function to any other image processing library written in C or C++, it is sufficient to (i) link (statically or dynamically) the miallib library and (ii) write a wrapper function to convert the original raster type to a miallib IMAGE, call segmentBinaryPatterns, and convert the returned miallib IMAGE back to the original raster type.

The effect of the 4 numerical input parameters on the resulting MSPA classification of a sample image are illustrated in Fig. 2. Upon calling segmentBinaryPatterns with the desired input parameters, a raster image containing the desired MSPA classification is returned.

2.2 Pre-processing

There are two pre-processing steps to handle no data and border effects respectively. No data values are handled by the function fm_preproc by creating a binary image with foreground pixels extended in the no data regions through a buffering proportional to the input size parameter. The buffering is efficiently computed by the miallib fast Euclidean distance transform following the algorithm proposed in (Meijster et al., 2000). The function fm_preproc returns a binary image with foreground pixels set to 1 and all other pixels set to 0.

Border effects are mitigated by extending the image definition domain with the addition of an image frame of width proportional to the input size parameter. The value of the pixels in the extended frame are set by propagating the values of the image border pixels of the input image in the direction matching their position. For example, left border pixels values are propagated in the extended left border along the right to left direction. This is achieved by the function fm_preproc2 that calls four times (one for each direction) the fast directional propagation algorithm.

Note that in case the internal input parameter is equal to 1, all holes of the binary image obtained by applying fm_preproc to the input image are extracted by filling its holes and performing the set difference between the latter and former images.
Perforations and edges are computed thanks to the function `setedges` that returns an array of two binary images, the first holding the edges and the second the perforations with foreground pixels (edges and perforations respectively) set to 1 and background pixels set to 0.

### 2.5 Islets

Islets (also called patches) are defined as those connected components of foreground pixels that do not contain any core pixels. They are obtained by subtracting the reconstruction by dilation\(^8\) of the foreground components using the core pixels as markers (seeds). The extraction of the islets is implemented within the function `getpatch` that returns a binary image with islet pixels set to 1 and all other pixels set to 0.

### 2.6 Connectors

Connectors link core-connected components. This is the more advanced feature of the morphological spatial pattern analysis because they cannot be extracted from local neighbourhood operations since connectors can be of any shape and length so that the full image extent needs to be considered at once. The key operation for the detection of connectors relies on morphological skeletonization with anchor points using a fast implementation based on FIFO queues (Iwanowski and Soille, 2005)\(^7\). The function `getconnector2core` details all necessary steps for the extraction of connectors. This function requires 3 input binary images: the first contains the image of core pixels, the second contains the morphological opening of the foreground pixels of the pre-processed input image, and the third image contains the residues defined as the foreground pixels of the pre-processed input image with the core, patch, perforations, and edges pixels set to 0 (obtained by successive subtraction operations):

\[
\text{IMAGE} \ast \text{getconnector2core (IMAGE} \ast \text{core, IMAGE} \ast \text{opening, IMAGE} \ast \text{residues, float size, int oitype, int graphfg, float edu)}
\]

Within the function `getconnector2core`, the skeletonisation with anchor points is computed on the union of the opening and residues images using the core pixels as anchor points. This operation generates a thinned (skeletonised) version of the target connectors. The latter are then obtained by performing a geodesic dilation of the skeleton using as geodesic mask the opening image to which the core image is subtracted and then the residues image added. This step is achieved within the function `getexternalboundarygeodesic` using the constrained (geodesic) Euclidean distance transform `ced`\(^9\) based on the algorithm described in (Soille, 1991). The function `getconnector2core` returns a binary image with all connectors set to 1 and all other pixels set to 0.

The connectors are categorised into two subclasses:

1. Bridges (also referred to as corridors): connector pixels emanating from two or more individual core-connected components;
2. Loops (also referred to as shortcuts): connector pixels emanating from the same core-connected component.

Bridges are obtained thanks to a combination of connected component labelling and a watershed algorithm from labelled markers\(^12\). They are obtained by calling the function `getcorridor` that requires the connector, core and, opening as input images:
The watershed function from markers \textit{wshab} (based on the algorithm described in Meyer and Beucher, 1990) is applied to the union of the opening and connectors with all pixels set to 255 and using the labelling of the core image as marker image. Those connectors that have all the same label in the output of the watershed from markers are defining the corridors since they connect core pixels having different labels. The function \textit{getcorridor} returns a binary image with all bridges set to 1 and all other pixels set to 0. Loops are defined by performing the set difference between connectors and bridges.

### 2.7 Branches

Pixels that do not belong to any of the previously defined categories are called branch pixels. They emanate from boundary pixels but do not provide any connections between core regions.

### 2.8 Background classes

The connected components of the background pixels of the input image are partitioned into 3 classes depending on their embedding relationship with the foreground pixel classes:

1. **Background:** refers to the connected components of background pixels of the input image that are connected to the image border. That is, they correspond to background pixels that can be reached from the image border by following a connected path of background pixels (i.e., without crossing any foreground pixels). They are obtained by considering the negation (i.e., complement) of the fillhole operation (see sec. 2.4) applied to the foreground pixels.

2. **Border-opening:** refers to the connected components of background pixels that cannot be reached from the image border without crossing one or more foreground pixels but that can be reached without crossing any core pixel. They are obtained by performing the intersection between the holes of the foreground and negation of the holes of the core pixels. Note that the holes of an image are simply obtained by filling its holes with the fillhole operation and then subtract the input image from the filled image. Border-opening are detected if and only if the input parameter internal is set to true (i.e., 1) and in this case the border-opening pixels are set to 220 in the output image.

3. **Core-opening:** refers to the connected components of background pixels that cannot be reached from the image border without crossing at least one core pixel. They are always associated to a perforation. In practice, all holes of image of core pixels are extracted and set to the value 100. In case the parameter internal is set to true (i.e., 1), the image with core holes set to 100 is added to the output image. This means that pixels belonging to core holes get their base label value plus an offset of 100. Accordingly, pixels in the output image with value 100 correspond to the core-opening pixels that are matching background pixels of the input image.

### 2.9 Output

The output image is defined with bit encoding, using bit shifting operations on each binary image containing the foreground classes described above and performing bitwise logical operations (or) between all the bit-shifted images. Note that in case the input parameter internal is equal to 1, embedded structures get a special code by adding an offset of 100 to the corresponding pixels while setting to 220 those holes of the input image that are not completely surrounded by perforation pixels (see details in Sec. 2.8). Finally, colours are assigned to each integer value of the output image using Red-Green-Blue colour coding stored in a look-up-table attached to the output raster image (the miallib raster image structure contains a field that can be used to store a look-up-table).

Table 1 indicates all possible MSPA classes for both foreground and background pixels as well as their respective byte value and corresponding colour coding (RGB value stored in the colour look-up-table referred to in the output miallib image of type IMAGE. Further information is available in the MSPA-Guide and applications to forest spatial patterns are outlined in (Vogt, 2022).

<table>
<thead>
<tr>
<th>Class</th>
<th>Color</th>
<th>RGB</th>
<th>Internal = 0</th>
<th>Internal = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Core</td>
<td>Green</td>
<td>000/200/000</td>
<td>17</td>
<td>17 / 117</td>
</tr>
<tr>
<td>2) Islet</td>
<td>Brown</td>
<td>160/060/000</td>
<td>9</td>
<td>9 / 109</td>
</tr>
<tr>
<td>3) Perforation</td>
<td>Blue</td>
<td>000/200/255</td>
<td>5</td>
<td>105</td>
</tr>
<tr>
<td>4) Edge</td>
<td>Black</td>
<td>000/000/000</td>
<td>3</td>
<td>3 / 103</td>
</tr>
<tr>
<td>5a) Loop</td>
<td>Blue</td>
<td>255/255/000</td>
<td>65</td>
<td>65 / 165</td>
</tr>
<tr>
<td>5b) Loop in Edge</td>
<td>Blue</td>
<td>255/255/000</td>
<td>67</td>
<td>67 / 167</td>
</tr>
<tr>
<td>5c) Loop in Perforation</td>
<td>Blue</td>
<td>255/255/000</td>
<td>69</td>
<td>69 / 169</td>
</tr>
<tr>
<td>5a) Bridge</td>
<td>Blue</td>
<td>255/000/000</td>
<td>33</td>
<td>33 / 133</td>
</tr>
<tr>
<td>6a) Bridge</td>
<td>Blue</td>
<td>255/000/000</td>
<td>35</td>
<td>35 / 135</td>
</tr>
<tr>
<td>6b) Bridge in Edge</td>
<td>Blue</td>
<td>255/000/000</td>
<td>37</td>
<td>37 / 137</td>
</tr>
<tr>
<td>7) Branch</td>
<td>Blue</td>
<td>255/160/000</td>
<td>1</td>
<td>1 / 101</td>
</tr>
<tr>
<td>Background</td>
<td>Blue</td>
<td>220/220/220</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Border-Opening</td>
<td>Blue</td>
<td>194/194/194</td>
<td>N/A</td>
<td>220</td>
</tr>
<tr>
<td>Core-Opening</td>
<td>Blue</td>
<td>136/136/136</td>
<td>N/A</td>
<td>100</td>
</tr>
<tr>
<td>No Data</td>
<td>Blue</td>
<td>255/255/255</td>
<td>129</td>
<td>129</td>
</tr>
</tbody>
</table>

Table 1. Class names, colour codes, and byte values featuring the MSPA classes stored in the image returned by segmentBinaryPatterns.

### 2.10 Library considerations

To minimise library dependencies, the Makefile of the miallib library contains a miallib\_mspa target that can be used to build a minimal set of the miallib C files required for segmentBinaryPatterns. The target name is miallib\_mspa.a. It is defined as a static library so that it can be statically linked to the target environment. This strategy is used for the distribution of MSPA in the GuidosToolbox (GTB) (Vogt and Riitters, 2017) and GuidosToolbox Workbench (GWB) (Vogt et al., 2022) that also contains an adaptation of mspa.c called fsp.c to accommodate two additional features, namely a dynamic timer (called loadBar) indicating the percentage of the overall execution and a Boolean parameter called disk indicating whether intermediate results should be written on disk to minimise the random access memory footprint so that large images can be processed.

### 3. PERFORMANCE

The performance of the algorithm is evaluated on images of increasing size as well as for on-the-fly computation for interactive analysis and exploratory visualisation. We demonstrate experimentally that the complexity of the proposed...
The proposed implementation handles images up to $2^{24}$ pixels. For example, a Global MSPA map of forest cover in equal area projection and with a pixel resolution of 100 meter (400,748 x 147,306 pixels, i.e. a 0.6 terapixel image) was processed on the JRC Big Data Analytics Platform\textsuperscript{18} (BDAP aka JEODPP) (Soille et al., 2018) in just 12 hours. Earlier examples of MSPA at European and Global extent can be browsed interactively as Google Earth image overlays\textsuperscript{19}. Processing large images is very much needed to mitigate dependencies with regards to the image definition domain. Indeed, pixel classes requiring the analysis of connectivity relations cannot be determined in a fixed local neighbourhood and may therefore depend on the observation domain in case some foreground connected components are extending beyond this domain.

Regarding the memory footprint of \texttt{segmentBinaryPatterns}, the peak RAM usage in bytes correspond to about 20 times the number of pixels of the input image (exact peak usage depends on image content for some of the underlying morphological image analysis algorithms). All these additional images are used to store the intermediate results as well as for temporary image buffers needed during the computations. When using the GTB adaptation that allows for writing intermediate results to disk (disk option in Sec. 2.10) and read them back during processing when needed rather than having them permanently in memory, the peak memory usage in bytes is reduced to 17 times the number of pixels of the input image.

Regarding the on-the-fly computation for interactive analysis and exploratory visualisation based on Jupyter notebooks\cite{DeMarchiAndSoille2019}, the proposed implementation is fast enough for integration in JupyterLab with on-the-fly computation in an area corresponding to the map view area and at resolution matching its zoom level. A Voil\’{a}\textsuperscript{20} dashboard based on a JupyterLab\textsuperscript{21} notebook is in preparation and will be referred to in the list of BDAP Voil\’{a} dashboards\textsuperscript{22}.

\section{4. Conclusions}

Morphological spatial pattern analysis has gained traction since its inception\cite{SoilleVogt2009} thanks to the accompanying MSPA website with extensive documentation, various GIS extensions, and a user-friendly provision of MSPA within the desktop application GTB and the server application GWB. With the provision of the MSPA source code under a free open source software license, we add yet another feature of software provisioning\cite{VogtRambaud2022} to increase the outreach of MSPA into the user community interested in pattern analysis. Since MSPA is available through a C library, it can be easily integrated in other data science environments. For instance, the MSPA function \texttt{segmentBinaryPatterns} is directly callable from the \texttt{pygeo} Python package\cite{Kempeneers2019} available at GitHub\textsuperscript{23} (see \texttt{segmentBinaryPatterns} method in \texttt{coopS} module\textsuperscript{24}) as well as from the morphological image analysis library wrapped to the XLISP-Plus Lisp interpreter\textsuperscript{25} (mialisp\textsuperscript{26}). We therefore expect the release of the MSPA code under an open source license to further boost its use for the analysis of geospatial patterns and indeed any other types of spatial patterns occurring in other scientific domains.

\section*{References}

Carlier, J., Davis, E., Ruas, S., Byrne, D., Caffrey, J. M., Coughlan, N. E., Dick, J. T., Lucy, F. E., 2020. Using open-

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image}
\caption{Top: The CPU time of MSPA of a map of the United States of America at different resolutions shows a linear relation with respect to the number of image pixels: measured values and estimations based on measured value at 100 m divided by $x^2$ where $x > 1$ is the multiple of 100, i.e., $x = 2, 3, \ldots, 10$. Bottom: MSPA output on the USA map resampled at 10,000 m resolution using mode resampling.}
\end{figure}


---

2 https://github.com/ec-jrc/jeolib-miallib
3 https://www.openmp.org/
4 https://github.com/ec-jrc/jeolib-miallib/blob/master/core/c/miallib.c
5 https://github.com/ec-jrc/jeolib-miallib/blob/071859cd3c78491e1928a0f746852f3f734df4bf4/ core/c/miallib.c#L433
6 https://github.com/ec-jrc/jeolib-miallib/blob/master/core/c/efedt.c
7 https://github.com/ec-jrc/jeolib-miallib/blob/57316294c73b190393347dc83a8371132c24fc5 0/core/c/ced.c#L94
8 https://github.com/ec-jrc/jeolib-miallib/blob/1f8b3df4a04d4018f70bfa7bee6e1f755476ac/ core/c/recons.c#L878
9 https://github.com/ec-jrc/jeolib-miallib/blob/1f8b3df4a04d4018f70b2bfa7bee6e1f755476ac/ core/c/skel.c#L587
10 https://github.com/ec-jrc/jeolib-miallib/blob/57316294c73b190393347dc83a8371132c24fc5 0/core/c/ced.c#L94
11 https://github.com/ec-jrc/jeolib-miallib/blob/1f8b3df4a04d4018f70b2bfa7bee6e1f755476ac/ core/c/wsfaah.c#L662
13 https://github.com/ec-jrc/jeolib-miallib/blob/main/external_sources/fsp/fsp.c
15 https://github.com/ec-jrc/GTB/blob/b24d0f3404bbc113380fa599fe4b912f0bb3f9/external_sources/fsp/fsp.c#L40
16 https://github.com/ec-jrc/GTB/blob/b24d0f3404bbc113380fa599fe4b912f0bb3f9/external_sources/fsp/fsp.c#L563
17 https://rasterio.readthedocs.io
18 https://jeodpp.jrc.ec.europa.eu/
20 https://github.com/voila-dashboards/voila
21 https://jupyter.org/
23 https://github.com/ec-jrc/jeolib-pyjeo
24 https://github.com/ec-jrc/jeolib-pyjeo/blob/573d919d07ae2af5bd20c8c744f6cface0a1e29/p yjeo/modules/ccops.py#L924
25 https://almy.us/xlisp.html
26 https://github.com/ec-jrc/jeolib-miallib/blob/071859cd3c78491e1928af0f746852f3724d4f8d4f/xlisp/c/xlglue1.c#L7798