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 Riitters, 2017), GuidosToolbox Workbench (GWB) (Vogt et al., 2022), and various GIS-extensions, see the MSPA home page[1] 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 GitHub[2]. Note that numerous functions of miallib support muti-threading based on OpenMP[3]. 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 GitHub[4].

2.1 Input and output

The miallib function implementing the morphological segmentation of binary patterns (Soille and Vogt, 2009) is named *segmentBinaryPatterns*[5]:

*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).

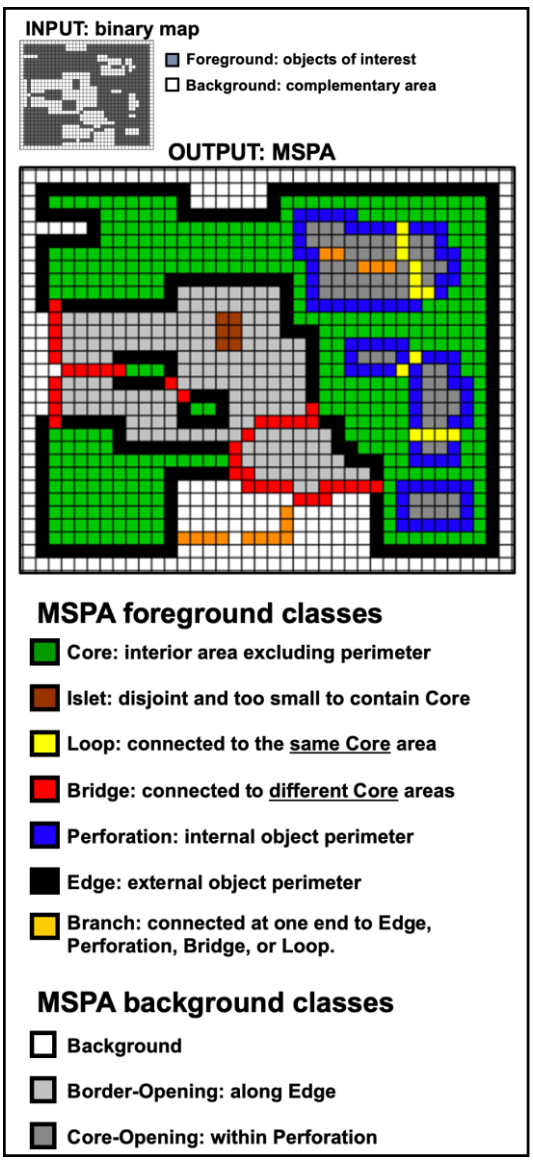


Figure 1. Morphological segmentation of binary patterns including the detection of connecting pathways leading to 7 classes for the foreground pixels and 3 classes for the background pixels.

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.

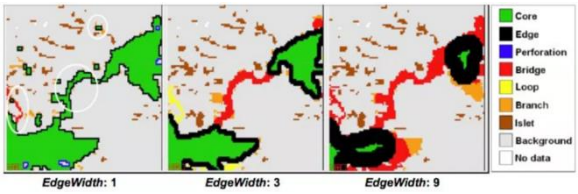
The effect of the 4 numerical input parameters on the resulting MSPA classification of a sample image are illustrated in Fig. 2. Upon calling *segmentBinaryPatterns* with the desired input parameters, a raster image containing the desired MSPA classification is returned.

2.2 Pre-processing

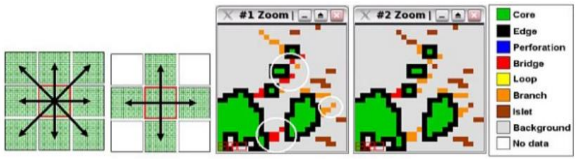
There are two pre-processing steps to handle no data and border effects respectively. No data values are handled by the function *fm_preproc* by creating a binary image with foreground pixels extended in the no data regions through a buffering proportional to the input size parameter. The buffering is efficiently computed by the miallib fast Euclidean distance transform[6] 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[7].

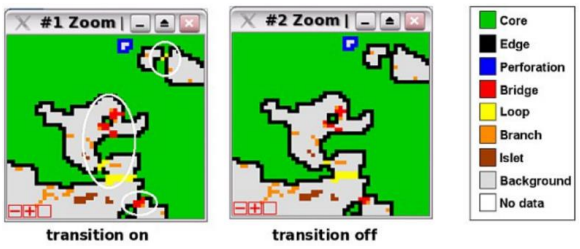
Note that in case the internal input parameter is equal to 1, all holes of the binary image obtained by applying *fm_preproc* to the input image are extracted by filling its holes and performing the set difference between the latter and former images.



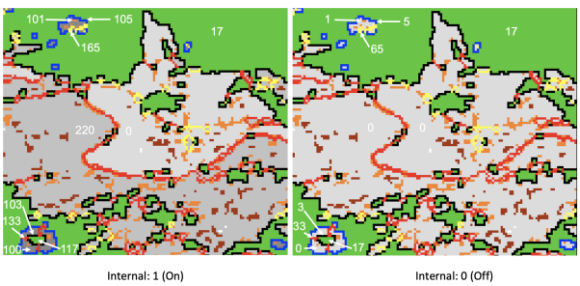
(a) effect of the parameter *size*



(b) effect of the parameter *graphfg* (i.e., connectivity rule)



(c) effect of the parameter *transition*



(d) effect of the parameter *internal*

Figure 2. The 4 MSPA input numerical parameters and their impact on the resulting segmentation.

2.3 Core

Core pixels are those pixels of the foreground connected components of the image that are lying far enough from their boundaries. They are obtained by considering all foreground pixels whose Euclidean distance transform lies beyond a threshold defined as the input size parameter. This is all implemented within the function *getcore* that returns a binary image with core pixels set to 1 and all other pixels set to 0.

2.4 Boundaries

Boundaries are defined as those foreground pixels that separate the core pixels from the background pixels. They are obtained by computing the external morphological gradient (Rivest et al., 1993) of the core pixels, that is, as the set difference between morphological dilation of the core pixels and the core pixels themselves. Boundary pixels are themselves divided into 2 categories called perforations and edges:

1. Perforations: they are defined as the inner boundary pixels. They are efficiently extracted thanks to the morphological fillhole operation (Soille and Ansoult, 1990) relying on the morphological reconstruction by erosion with a fast algorithm based on FIFO queues[8].

2. Edges: they are defined as the outer boundary pixels. They are obtained by subtracting the perforations from the boundaries. A float number greater or equal to 1 indicating the width of the edges.

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[9] 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 skeletonisation with anchor points using a fast implementation based on FIFO queues (Iwanowski and Soille, 2005)[10]. 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):

*IMAGE *getconnector2core (IMAGE *core, IMAGE * opening, IMAGE *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*[11] 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:

*IMAGE *getcorridor (IMAGE *connector, IMAGE *core, IMAGE *opening, float size, int oitype, int graph)*

The watershed function from markers *wsfah* (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 *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[13] and applications to forest spatial patterns are outlined in (Vogt, 2022).

Class	Color	RGB	Internal = 0	Internal = 1
1) Core		000/200/000	17	17 / 117
2) Islet		160/060/000	9	9 / 109
3) Perforation		000/000/255	5	105
4) Edge		000/000/000	3	3 / 103
5a) Loop		255/255/000	65	65 / 165
5b) Loop in Edge		255/255/000	67	67 / 167
5c) Loop in Perforation		255/255/000	69	69 / 169
6a) Bridge		255/000/000	33	33 / 133
6b) Bridge in Edge		255/000/000	35	35 /135
6c) Bridge in Perforation		255/000/000	37	37 / 137
7) Branch		255/140/000	1	1 / 101
Background		220/220/220	0	0
Border-Opening		194/194/194	N/A	220
Core-Opening		136/136/136	N/A	100
No Data		255/255/255	129	129

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[14] to accommodate two additional features, namely a dynamic timer (called loadBar[15]) indicating the percentage of the overall execution and a Boolean parameter called disk[16] 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

implementation is in *O(n)*. That is, the computational time increases linearly with the number of pixels *n* of the input image. This is illustrated in Fig. 3. The input image is a GeoTIFF file at 100 m resolution with a bounding box matching that of the conterminous USA and Alaska and forest pixels with value 2, non-forest pixels with value 1, and no data pixels with value 0. This image has been resampled with GDAL to multiples of 100 m up to 1000 m using mode sampling with the rasterio package[17] providing GDAL python bindings. CPU time for each resolution was measured using the function *process_time* of the Python time package. The CPU time for all images with resolution that are multiple of 100 m was estimated as measured CPU time for the 100 m resolution image divided by x^2 where $x > 1$ is the multiple of 100, i.e., $x = 2, 3, ..., 10$, as one would expect for a computational complexity that is linear with the number of image pixels. The excellent match between the measured and estimated CPU time demonstrates experimentally that the complexity of the proposed open source release of MSPA is indeed linear. Note that in this experiment, the input parameters are 8 for connectivity (graphfg) and 1 for the parameters size, internal, and transition. All calculations were conducted on an Intel(R) Xeon(R) CPU E7-8870 v4 at 2.10GHz and the environment variable *OMP_NUM_THREADS* was set to 1 for the experiment so that no multi-threading was considered.

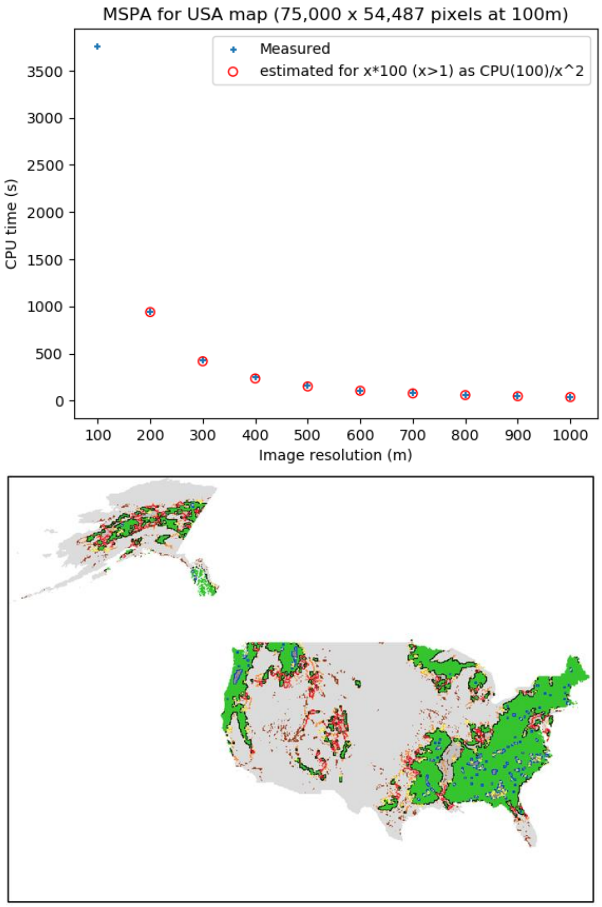


Figure 3. Top: The CPU time of MSPA of a map of the United States of America at different resolutions shows a linear relation with respect to the number of image pixels: measured values and estimations based on measured value at 100 m divided by x^2 where $x > 1$ is the multiple of 100, i.e., $x = 2, 3, ..., 10$. Bottom: MSPA output on the USA map resampled at 10,000 m resolution using mode resampling.

The proposed implementation handles images up to 2^{64} 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[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[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 *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 (De Marchi and Soille, 2019), 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 Voilà[20] dashboard based on a JupyterLab[21] notebook is in preparation and will be referred to in the list of BDAP Voilà dashboards[22].

4. CONCLUSIONS

Morphological spatial pattern analysis has gained traction since its inception (Soille and Vogt, 2009) 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 (Vogt and Rambaud, 2022) 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 *segmentBinaryPatterns* is directly callable from the *pyjeo* Python package (Kempeneers et al., 2019) available at GitHub[23] (see *segmentBinaryPatterns* method in *ccops* module[24]) as well as from the morphological image analysis library wrapped to the XLISP-Plus Lisp interpreter[25] (mialisp[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.

REFERENCES

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

source software and digital imagery to efficiently and objectively quantify cover density of an invasive alien plant species. *Journal of Environmental Management*, 266, 110519. doi.org/10.1016/j.jenvman.2020.110519.

De Marchi, D., Soille, P., 2019. Advances in interactive processing and visualisation with JupyterLab on the JRC Big Data Platform (JEODPP). *Proc. of the 2019 conference on Big Data from Space (BiDS'19)*, Publications Office of the European Union, Luxembourg, 45–48. doi.org/10.5281/zenodo.3239239.

Iwanowski, M., Soille, P., 2005. A queue based algorithm for order independent anchored skeletonisation. *Lecture Notes in Computer Science*, 3691, 530-537. doi.org/10.1007/11556121_65.

Kempeneers, P., Pesek, O., De Marchi, D., Soille, P., 2019. A Python Package For The Analysis of Geospatial Data. *International Journal of Geo-Information*, 8(10). doi.org/10.3390/ijgi8100461.

Meijster, A., Roerdink, J., Hesselink, W., 2000. A general algorithm for computing distance transforms in linear time. J. Goutsias, L. Vincent, D. Bloomberg (eds), *Mathematical Morphology and its Applications to Image and Signal Processing*, Computational Imaging and Vision, 18, Kluwer Academic Publishers, Boston, 331–340. Proc. of ISMM'2000, Palo Alto, June 26–29. doi.org/10.1007/0-306-47025-X_36.

Meyer, F., Beucher, S., 1990. Morphological segmentation. *Journal of Visual Communication and Image Representation*, 1(1), 21-46. doi.org/10.1016/1047-3203(90)90014-M.

Modica, G., Praticò, S., Laudari, L., Ledda, A., Di Fazio, S., De Montis, A., 2021. Implementation of multispecies ecological networks at the regional scale: analysis and multi-temporal assessment. *Journal of Environmental Management*, 289, 112494. doi.org/10.1016/j.jenvman.2021.112494.

Ossola, A., Locke, D., Lin, B., Minor, E., 2019. Yards increase forest connectivity in urban landscapes. *Landscape Ecology*, 34(12). doi.org/10.1007/s10980-019-00923-7.

Rincón, V., Velázquez, J., Gutiérrez, J., Hernando, A., Khoroshev, A., Gómez, I., Herráez, F., Sánchez, B., Pablo Luque, J., García-abril, A., Santamaría, T., Sánchez, D.-M., 2021. Proposal of new Natura 2000 network boundaries in Spain based on the value of importance for biodiversity and connectivity analysis for its improvement. *Ecological Indicators*, 129, 108024. doi.org/10.1016/j.ecolind.2021.108024.

Rivest, J.-F., Soille, P., Beucher, S., 1993. Morphological gradients. *Journal of Electronic Imaging*, 2(4), 326-336. doi.org/10.1117/12.159642.

Soille, P., 1991. Spatial distributions from contour lines: an efficient methodology based on distance transformations. *Journal of Visual Communication and Image Representation*, 2(2), 138-150. doi.org/10.1016/1047-3203(91)90004-Y.

Soille, P., 2004. *Morphological Image Analysis: Principles and Applications*. corrected 2nd printing of the 2nd edn, Springer-Verlag, Berlin and New York. doi.org/10.1007/978-3-662-05088-0.

Soille, P., Ansoult, M., 1990. Automated basin delineation from Digital Elevation Models using mathematical morphology. *Signal Processing*, 20, 171-182. doi.org/10.1007/11556121_65.

Soille, P., Burger, A., De Marchi, D., Kempeneers, P., Rodriguez, D., Syrris, V., Vasilev, V., 2018. A versatile data-intensive computing platform for information retrieval from big geospatial data. *Future Generation Computer Systems*. 81: 30–40, doi.org/10.1016/j.future.2017.11.007.

Soille, P., Vogt, P., 2009. Morphological segmentation of binary patterns. *Pattern Recognition Letters*, 30(4), 456-459. doi.org/10.1016/j.patrec.2008.10.015.

Vogt, P., 2022. Measuring forest spatial pattern with mathematical morphology. Online. https://ies-ows.jrc.ec.europa.eu/gtb/GTB/psheets/GTB-Pattern-Morphology.pdf

Vogt, P., Rambaud, P., 2022. Tackling the challenges of software provision. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, FOSS4G 2022 — Academic Track, This volume.

Vogt, P., Riitters K., 2017. GuidosToolbox: universal digital image object analysis. *European Journal of Remote Sensing*, 50, 1, pp. 352-361, doi.org/10.1080/22797254.2017.1330650.

Vogt, P., Riitters, K., Rambaud, P., d'Annunzio, R., Lindquist, E., Pekkarinen, A., 2022. GuidosToolbox Workbench: spatial analysis of raster maps for ecological applications. *Ecography*, doi.org/10.1111/ecog.05864.

[1] https://forest.jrc.ec.europa.eu/en/activities/lpa/mspa/
[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/mspa.c
[5] https://github.com/ec-jrc/jeolib-miallib/blob/071859cd3c78491e1928a0f746852f3724d4f84f/core/c/mspa.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/57316294c73b190393347dc83a8371132c24fc50/core/c/imstat.c#L2047
[8] https://github.com/ec-jrc/jeolib-miallib/blob/1f8b3df4a04d4018f70b2bfaaeb6e1f755476acb/core/c/recons.c#L878
[9] https://github.com/ec-jrc/jeolib-miallib/blob/1f8b3df4a04d4018f70b2bfaaeb6e1f755476acb/core/c/recons.c#L483
[10] https://github.com/ec-jrc/jeolib-miallib/blob/1f8b3df4a04d4018f70b2bfaaeb6e1f755476acb/core/c/skel.c#L587
[11] https://github.com/ec-jrc/jeolib-miallib/blob/57316294c73b190393347dc83a8371132c24fc50/core/c/ced.c#L94
[12] https://github.com/ec-jrc/jeolib-miallib/blob/1f8b3df4a04d4018f70b2bfaaeb6e1f755476acb/core/c/wsfah.c#L669
[13] https://ies-ows.jrc.ec.europa.eu/gtb/GTB/MSPA_Guide.pdf
[14] https://github.com/ec-jrc/GTB/blob/main/external_sources/fsp/fsp.c

[15] https://github.com/ec-jrc/GTB/blob/b24d0f3404bbc113380fa599fe4b912f0bcb3f93/external_sources/fsp/fsp.c#L40
[16] https://github.com/ec-jrc/GTB/blob/b24d0f3404bbc113380fa599fe4b912f0bcb3f93/external_sources/fsp/fsp.c#L563
[17] https://rasterio.readthedocs.io
[18] https://jeodpp.jrc.ec.europa.eu/
[19]https://forest.jrc.ec.europa.eu/en/activities/lpa/gtb/#GEoverlays
[20] https://github.com/voila-dashboards/voila
[21] https://jupyter.org/
[22] https://jeodpp.jrc.ec.europa.eu/bdap/voila/
[23] https://github.com/ec-jrc/jeolib-pyjeo
[24] https://github.com/ec-jrc/jeolib-pyjeo/blob/573d919d07ae2af5bd20c8c744fb6cfacc0a1e29/pyjeo/modules/ccops.py#L924
[25] https://almy.us/xlisp.html
[26] https://github.com/ec-jrc/jeolib-miallib/blob/071859cd3c78491e1928a0f746852f3724d4f84f/xlisp/c/xlglue1.c#L7798