GWB_SPATCON

Kurt Riitters, Peter Vogt, January 2023.

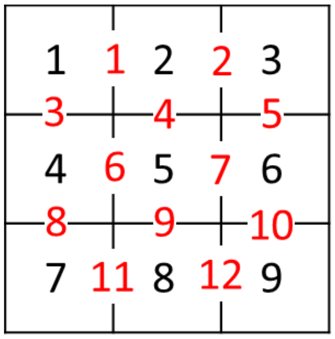
Supplemental documentation of features, options, and use of the spatial convolution program ***spatcon.*** The latest version of this document, and the most recent spatcon source code, is available on GitHub.

Contents.

1. General description.

Spatcon implements a moving window analysis of a two-dimensional image (or a map) comprised of pixels (grid cells). It moves a square window across the image, one pixel at a time, accumulating and discarding information along the way. Based on the pixel values in the window on the input image at a given pixel location, a metric is calculated and assigned to that location on the output image. Thus, the output pixel value codes the input context of that pixel location, and the spatial resolution of the input image is preserved. The spatial scale of the analysis is defined by the size of the window[1]. All *spatcon* metrics are calculated from either (a) the frequencies of different input pixel values in the window, or (b) the pixel value adjacency matrix for the window, which contains the frequencies of different types of pixel value adjacencies in the window. Adjacency is defined by the 4-neighbor rule, and adjacencies at the boundary of the window are ignored. For example, a 3x3 window contains 9 pixels and 12 adjacencies (Fig. 1).

Figure 1. Example 3x3 pixel window containing 9 pixels (numbered in black font) and 12 adjacencies (red font).



[1] Other aspects of scale are defined by the input dataset – spatial resolution (pixel size), spatial extent (image size), and thematic resolution (image legend).

2. Important notes: missing values, re-coding input pixel values, output images.

2.1. Missing values.

In *spatcon*, the input image is "buffered" on all four sides with missing pixel values, where the buffer size is selected so that a full window can be centered on every pixel in the input image[2]. User-defined missing pixels can also occur within the image area. The metrics are calculated based on whatever is in a window, and the user chooses to include or ignore missing values. The result is stored as the location of the output pixel in the center of the window, even if the user has defined that input pixel as missing. For example, suppose that ocean water is defined as missing on an input land cover image. The pattern metric image may have metric values where the ocean water is near the coastline because a portion of the window is not ocean water.

Spatcon always produces a missing value if there is no information in the window to perform a given calculation. For metrics calculated from pixel value frequencies, this can happen when all pixels in the window are missing. For metrics calculated from the frequencies of pixel value adjacencies, this can happen when all requested adjacencies within the window are missing; an adjacency is missing if either of the two adjacent pixels is missing. Thus, it is possible for a window which contains a given pixel value to have no adjacencies for that pixel value, for example a checkerboard image with only one pixel value separated by missing pixels. Adjacencies at the boundary of a window are ignored, i.e., they are not treated as missing data.

While the user may choose to include or ignore missing values in metric calculations, it is recommended to ignore them unless there is a compelling reason to include them. If they are included, then it may be difficult to interpret some of the metrics (see section 4). This is because the *spatcon* INTERNAL pixel value for missing data is ALWAYS zero. That is, the user may define a non-zero input pixel value to be missing, but that value is converted to an internal zero for metric calculations.

2.2. Re-coding pixels.

If requested, *spatcon* will re-code the input pixel values using a user-specified lookup table. The *spatcon* parameters m, a, and b must refer to input pixel values AFTER the requested re-coding.

2.3. Output maps.

In an 8-bit output map, the missing value is always 0 (zero). In a 32-bit output map, the missing value is always -0.01.

The default output format is 8-bit (byte). However, many of the metrics are computed as float values, and the re-scaling to byte values in the range [1,255] is done as byte value = (float value * 254) + 1. Note that a true zero metric value is stored as the number 1, reserving metric value zero for missing values. The user may re-scale byte output to float output in other software: float value = (byte value – 1) / 254 (or simply select float output from *spatcon*).

---

[2] If w is the side length of the window, the buffer width is $(w\text{-}1)/2$ pixels. For a pixel at the corner of a rectangular image, at least ¾ of the window is in the buffer area. The buffer area is removed before writing the output image.

2.4. Hardware requirement.

Available RAM (bytes)[3]: For byte output (see below), approximately 2 X (number of rows) X (number of columns). For float output (see below), approximately 5 X (number of rows) X (number of columns).

On multi-core machines, all cores that are available within the executing shell will be used (default). To use fewer cores, perform the optional environment setting (Section 5) before executing GWB_SPATCON.

3. Using GWB_SPATCON: parameter specification, optional re-code file format.

3.1. Parameter specification.

The spatcon parameters are specified in the GWB_SPATCON file “spatcon-parameters.txt”. Each line has a parameter code (a letter), a space, and a parameter value (a number). Parameters that are not needed for the selected metric are ignored. Important: The spatcon parameters m, a, and b refer to values AFTER optional re-coding.

Required:

- r or R = mapping rule. Permitted values: 1, 6, 7, 10, 20, 21, 51, 52, 53, 54, 71, 72, 73, 74, 75, 76, 77, 78, 81, 82, 83
- w or W = window size. the window will be a square with side length w pixels. Must be an odd number (3, 5, 7...). Minimum = 3 (i.e., 3 pixels X 3 pixels). The maximum must be less than the smaller of (number of rows, number of columns) in the input image.

Required for some mapping rules:

- a or A = "First target code" required for mapping rules 75, 76, 77, 78, 81, 82, 83. Default = 0.
- b or B = "Second target code" required for mapping rules 76, 78, 82, 83. Default = 0.

Optional:

- h or H = How missing pixels or adjacencies are handled during metric calculation. 1 = Ignore (default). 2 = Include in calculations. Option ‘2’ is not available for mapping rules 21, 82, 83.
- f or F = Output precision. 0 = 8-bit byte (default). 1 = 32-bit float. Float is not available for mapping rules 1, 6, 7, 10.
- z or Z = Request re-code of input pixels. 0 = No (default). 1 = Yes
- m or M = missing value. This specifies which pixel value is missing, AFTER optional re-coding. Default = 0.

3.2. Re-coding input pixel values (optional)

The re-coding parameters are specified using the GWB_RECODE file “rec_parameters.txt”.

[3] It is strongly recommended to NOT use swap space for large images because disk I/O speed is a major constraint. Section 7 describes converting the input image into overlapping tiles. For this, the overlap width should at least one-half the largest window size that will be used; the overlap area can be removed later during image un-tiling.

4. *Spatcon* metrics.

Below, “Rx” refers to “Rule x” (metric number x), which is selected by the “r” parameter.

- **R1. Majority (most frequent) pixel value.**
  Description: the most frequent pixel value in the window.
  Notes:
  1. How are ties handled? During image input, the pixel values are re-coded for internal storage, and this depends on the order in which a new pixel value is encountered on the input image. If there is a tie for the majority, the lowest re-coded pixel value is returned.
  2. Float output not available.

- **R6. Landscape mosaic (19-class version)**, and
- **R7. Landscape mosaic (103-class version)**
  Description: classification of the window based on the relative frequency of three different pixel values.
  Notes:
  1. The classification model is a tri-polar (ternary) chart, with the three axes corresponding to the proportions of input pixel values 1, 2, and 3. See Peter Vogt’s product sheet for additional description and examples.
  2. The non-missing pixels in the image (after optional re-coding) must be in 1, 2, 3 only.
  3. Float output not available.

- **R10. Number of unique pixel values.**
  Description: the number of unique pixel values in the window.
  Notes:
  1. Float output not available.

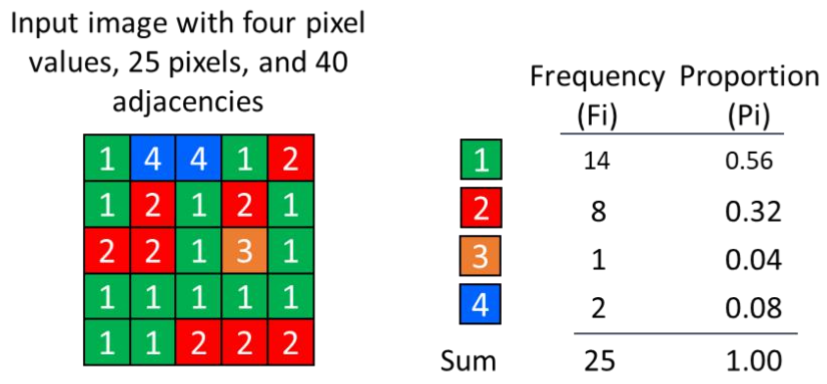
- **R20. Median pixel value.**
  Description: the median pixel value in the window.
  Notes:
  1. Calculated by constructing the cumulative distribution of pixel values. The median pixel value is the one for which the cumulative number of pixels first exceeds 50% of the total number of pixels.
  2. If missing values are included in the calculation, they are represented by internal pixel value 0 (zero), which complicates interpretation.
  3. This metric is not interpretable for categorical input data; input data should be ordinal or numeric.
  4. While float output is available, the result is always an integer value because of the way the median is calculated, and byte output is recommended.
  5. An input pixel value of 0 is assumed to be missing data[4].

[4] This can be awkward if zero is supposed to represent a real value of zero, instead of being treated as missing data. The user must rescale the input data, e.g., from [0, 100] to [1, 100], and assign pixel value zero to “true missing” pixels.

- **R21. Mean pixel value.**
  Description: the mean (arithmetic average) pixel value in the window.
  Notes:
  1. Calculated as the sum of pixel values divided by the number of pixels, rounded to the nearest byte value.
  2. Missing values are always excluded; the parameter “h” has no effect.
  3. This metric is not interpretable for categorical input data; input data should be ordinal or numeric.
  4. An input pixel value of 0 is assumed to be missing data (see footnote to R20).

- **R51. Gini-Simpson pixel diversity.**
  Description: measures pixel value diversity in the window; larger numbers indicate lower diversity.
  Notes:
  1. The Gini-Simpson index equals one minus Simpson’s classical index; larger values indicate lower diversity.
  2. Let Pi be the proportion of the i-th pixel value (Fig. 2). The index is calculated as (1 – ∑(Pi*Pi)).
  3. May be difficult to interpret when missing values are included.

Figure 2. Illustration of the tabulation of Fi and calculation of Pi on an input image with four unique pixel values (N = 4).


Input image with four pixel values, 25 pixels, and 40 adjacencies
1 4 4 1 2
1 2 1 2 1
2 2 1 3 1
1 1 1 1 1
1 1 2 2 2

	Frequency (Fi)	Proportion (Pi)
1	14	0.56
2	8	0.32
3	1	0.04
4	2	0.08
Sum	25	1.00

- **R52. Gini-Simpson pixel evenness.**
  Description: measures pixel value evenness in the window; larger numbers indicate higher evenness.
  Notes:
  1. Calculated as the observed Gini-Simpson pixel diversity index divided by the maximum Gini-Simpson pixel diversity index, where the maximum index depends on the number of unique pixel values.
  2. Let Pi be the proportion of the i-th pixel value (Fig. 2), and let N be the number of unique pixel values. The maximum Gini-Simpson index, obtained when all Pi are the same value, equals 1 – 1/N. The metric is calculated as (1 – ∑(Pi*Pi))/ (1 – 1/N). The potential divide-by-zero error for the case of only one pixel value is avoided by setting the index in that case to the maximum (1.0 float, 255 byte).
  3. May be difficult to interpret when missing values are included.

- **R53. Shannon pixel evenness.**
  Description: measures pixel value evenness in the window; larger numbers indicate higher evenness.
  Notes:
  1. Calculated as the classical Shannon diversity index divided by the maximum Shannon index, where the maximum depends on the number of unique pixel values.
  2. Let Pi be the proportion of the i-th pixel value (Fig. 2), and let N be the number of unique pixel values. Shannon diversity equals –∑(Pi*log(Pi)), and larger values indicate lower diversity (hence, higher evenness). The maximum diversity, obtained when all Pi are the same, equals log(N). The metric is calculated as –∑(Pi*log(Pi)) / log(N). The potential divide-by-zero error for the case of only one pixel value is avoided by setting the index in that case to the maximum (1.0 float, 255 byte).
  3. May be difficult to interpret when missing values are included.

- **R54. Pmax.**
  Description: The proportion of the total number of pixels having the most frequent pixel value.
  Notes:
  1. The proportion of each unique pixel value is computed (Fig. 1), and Pmax is the maximum of those proportions.

Note for adjacency metrics R71 - R78:
These metrics are derived from the attribute adjacency matrix in the window. Let M be a t X t adjacency matrix where t is the number of unique pixel values, and the cells of M contain Fij, the frequency of pixel adjacencies of type i|j (i.e., a pixel of value i adjacent to a pixel of value j) within a window. Window boundary adjacencies are excluded. (Fig. 1).

Spatcon provides several metrics related to the classical "contagion" metric from the landscape ecology literature, which was first developed to measure the tendency for individual pixel values to appear "clumped together" on an image. These classical metrics typically preserved the order of pixels in each pair, such that i|j was tabulated separately from the pair j|i (Fig. 3). For consistency with the literature at that classical time, the *spatcon* algorithms for R71, 72, and 73 preserve pixel order. It must be noted, however, that the order for a given pair of pixels arbitrarily depends on the adjacency rule by which a set of pixel pairs is tabulated. Riitters et al. (1996) investigated this dependence on pixel order (and other aspects of constructing the attribute adjacency matrix). They concluded that the classical contagion metrics measure the entropy of the entire adjacency matrix (i.e., not specifically the lack of entropy due to clumping alone[5]), and proposed R74 as an alternative metric that is more robust to clumping alone.

Pixel order is not relevant for *spatcon* metrics R75, R76, R77, and R78 which describe one or two specific types of pixel adjacencies. For these metrics, the adjacency i|j to equivalent to the adjacency j|i (Fig. 4). For i ≠ j, the total number of adjacencies between pixel values 1 and 2 equals Fi,j_j + Fj,i. For i = j, the total number is Fi,i. In the following examples for these metrics (Figs. 5 – 8), the "first target code" is "1" for clarity of presentation, but the "first target code" can be any valid pixel value.

[5] In the attribute adjacency matrix, entropy can be lower if the diagonal elements are large (due to tendency for a given pixel value to be adjacent to itself, i.e., clumping), or if the non-diagonal elements are large (due to tendency for two different pixel values to be adjacent.

Figure 3. Illustration of the tabulation of Fij and calculation of Pij on the input image (Fig. 2) with four unique pixel values (t = 4), preserving the order of pixel values in adjacent pixel pair, for metrics R71, R72, R73. In this example, the adjacency rule is left|right and top|bottom, thus tabulating each adjacency only one time. Metric R74 is the sum of the proportions on the diagonal.

Frequency (Fij)

	1	2	3	4	Sum
1	12	9	1	1	
2	6	4	1	0	
3	2	0	0	0	
4	2	1	0	1	
Sum					40

Proportion (Pij)

	1	2	3	4	Sum
1	0.300	0.225	0.025	0.025	
2	0.150	0.100	0.025	0	
3	0.050	0	0	0	
4	0.050	0.025	0	0.025	
Sum					1.000

Figure 4. Illustration showing the adjacency matrix for the input image (Fig. 2) when not preserving the order of pixels in pairs (c.f., Fig. 3). This form of the adjacency matrix is used for R75, R76, R77, and R78.

	1	2	3	4	Sum
1	12	15	3	3	
2	--	4	1	1	
3	--	--	0	0	
4	--	--	--	1	
Sum					40

- **R71. Angular second moment**

Description: Haralick's classical texture metric.
Notes:
1. Haralick's metric is essentially Simpson's diversity index (c.f., R51) applied to frequencies of pixel value adjacencies instead of the frequencies of pixel values.
2. Let Pij be the proportion of total adjacencies which are of adjacency type i|j (Fig. 3). Angular second moment equals $\sum\sum(Pij*Pij)$.
3. May be difficult to interpret when missing values are included.

- **R72. Gini-Simpson adjacency evenness.**

Description: Analogous to Gini-Simpson's pixel evenness index (R52), measures the evenness of the adjacency frequencies.
Notes:
1. Calculated as an observed index divided by the maximum index, where the maximum index depends on the number of possible types of adjacencies, which depends on the number of unique pixel values in the window.
2. Let Pij be the proportion of total adjacencies which are of adjacency type i|j (Fig. 3), and let t be the number of unique pixel values in the window. The observed index equals $1 - \sum\sum(Pij*Pij)$ (c.f., R71). The maximum index, obtained when all $Pij = 1/t^2$, equals $1 – (1 / (t^2))$. The metric is calculated as $1 – [[1 - \sum\sum(Pij*Pij)] / [1 – (1 / (t^2))]]$.
3. May be difficult to interpret when missing values are included.

- **R73. Shannon adjacency evenness**

Description: "Classical" contagion in the landscape ecology literature, derived by O'Neill et al (1988) with a correction by Li and Reynolds (1993), is analogous to Shannon's pixel evenness index (R53).
Notes:
1. Calculated as an observed index divided by the maximum index, where the maximum index depends on the number of possible types of adjacencies, which depends on the number of unique pixel values in a window.
2. Let Pij be the proportion of total adjacencies which are of adjacency type i|j (Fig. 3). Let t be the number of unique pixel values. The observed index equals $-1 * \sum\sum Pij*\log(Pij)$. The maximum index value, obtained when all $Pij = 1/t^2$, equals $2*\log(t)$. The metric is calculated as $1 – [[-1 * \sum\sum Pij*\log(Pij)] / [2*\log(t)]]$.
2. Larger values indicate less entropy in the attribute adjacency matrix, which may be related to clumping.
3. May be difficult to interpret when missing values are included.
4. The potential error in the logarithm function is avoided by assuming $\log(0) = 0$.

- **R74. Sum of diagonals**

Description: a measure of overall contagion (clumping) considering all pixel values in a window.
Notes:
1. Calculated as the total number of pixel-value adjacencies that are self-adjacencies, divided by the total number of adjacencies. Let Pii be the proportion of total adjacencies which are of type i|i (Fig. 3). The metric is calculated as $\sum Pii$.
2. Larger values indicate more clumping over all pixel values in the window.
3. May be difficult to interpret when missing values are included.

- **R75. Proportion of total adjacencies involving a specific pixel value.**

Description. The relative frequency of pixel adjacencies involving a specific pixel value.
Notes:
1. Calculated for the specified pixel value as the sum of adjacencies involving the specific pixel value divided by the total number of adjacencies in a window (Fig. 5).
2. The specific pixel value is selected by the input parameter “first target code”.
3. In this metric, the adjacency could be an adjacency between pixels with the same pixel value.

Figure 5. Illustration of the calculation of R75 for the adjacency matrix in Fig. 4.

	1	2	3	4	Sum
1	12	15	3	3	33
2	--	4	1	1	
3	--	--	0	0	
4	--	--	--	1	
Sum					40

R75. Proportion of total adjacencies involving a specific pixel value.
First target code = 1.
Calculation:
33 / 40 = 0.825

- **R76. Proportion of total adjacencies which are between two specific pixel values.**

Description: The relative frequency of a specific type of pixel value adjacency in the window.
Notes:
1. Metric is calculated as the frequency of adjacencies for two specific pixel values, divided by the total number of adjacencies (Fig. 6).
2. The two specific byte values are selected by the input parameters “first target code” and “second target code”. The order of the two target codes is not important
3. The two target codes may be the same value (Fig. 6).

Figure 6. Illustration of the calculation of R76 for the adjacency matrix in Fig. 4.

	1	2	3	4	Sum
1	12	15	3	3	33
2	--	4	1	1	
3	--	--	0	0	
4	--	--	--	1	
Sum					40

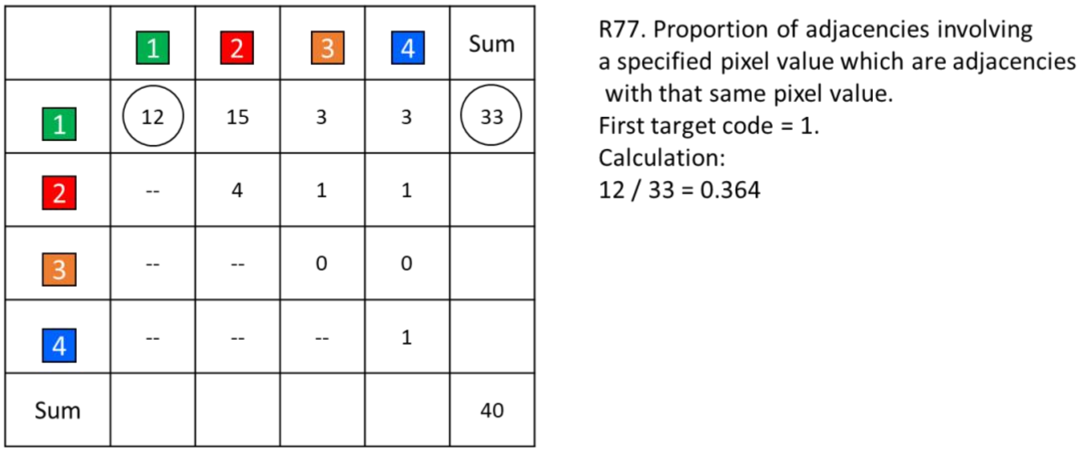
R76. Proportion of total adjacencies which are between two specific pixel values.
First target code = 1.
Second target code = 3.
Calculation:
3 / 40 = 0.075

If the first and second target codes = 1,
The calculation is 12/40 = 0.300

- **R77. Proportion of adjacencies involving a specified pixel value which are adjacencies with that same pixel value.**

Description: Describes the contagion or clumping of the specified pixel value.
Notes:
1. The metric is calculated for the specified pixel value as the number of same value adjacencies divided by the total number of adjacencies involving that pixel value (Fig. 7).
2. The value indicates pixel value “contagion.”
3. The specific pixel value is selected by the input parameter “first target code.”
4. If “fragmentation” is the opposite of “contagion,” then the metric complement (1 - R77) indicates “fragmentation” of the specific pixel value, which is relevant when using R78.

Figure 7. Illustration of the calculation of R77 for the adjacency matrix in Fig. 4.



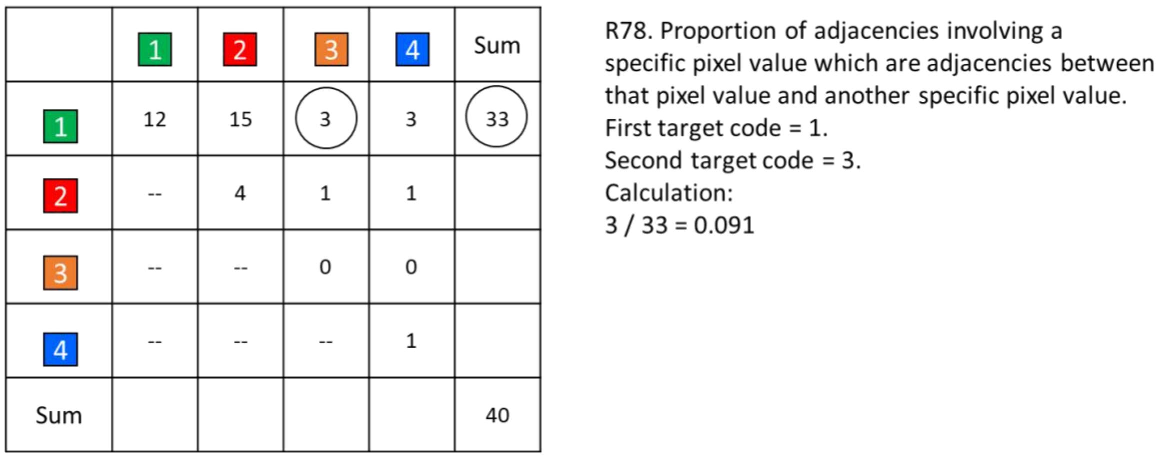
	1	2	3	4	Sum
1	12	15	3	3	33
2	--	4	1	1	
3	--	--	0	0	
4	--	--	--	1	
Sum					40

R77. Proportion of adjacencies involving
a specified pixel value which are adjacencies
with that same pixel value.
First target code = 1.
Calculation:
12 / 33 = 0.364

- **R78. Proportion of adjacencies involving a specific pixel value which are adjacencies between that pixel value and another specific pixel value.**

Description: The relative contribution of the second pixel value (“second target code”) to the adjacencies involving the first pixel value (“first target code”) (c.f., R77).
Notes:
1. The metric is calculated for the two specified pixel values as the number of pixel adjacencies of that type, divided by the total number of adjacencies involving the first pixel value (Fig. 8).
2. Repeating this metric for other selections of the second pixel value allows partitioning the “fragmentation” of the first pixel value (Wade et al. 2003).
3. The specific pixel values are selected by the input parameters “first target code” and “second target code”. The order of target codes is important. If the two target codes are the same, the result is the same as for R77.

Figure 8. Illustration of the calculation of R78 for the adjacency matrix in Fig. 4.



	1	2	3	4	Sum
1	12	15	3	3	33
2	--	4	1	1	
3	--	--	0	0	
4	--	--	--	1	
Sum					40

R78. Proportion of adjacencies involving a specific pixel value which are adjacencies between that pixel value and another specific pixel value.
First target code = 1.
Second target code = 3.
Calculation:
3 / 33 = 0.091

The remaining metrics are based on the frequencies of different pixel values (Fig. 2).

- **R81. Area density.**

Description: the proportion of a window that is a specific pixel value.
Notes:
1. Calculated as the number of pixels of “first target code” divided by the total number of pixels in a window.
2. Interpretation as fragmentation: if the specified pixel value was not fragmented, then R81 = 1.0, and departures from that baseline indicate fragmentation.
3. It is arguably the basic measure of fragmentation because all other measures of fragmentation (e.g., 1 - R77) depend on area density; the simplest example is that if R81 = 1 then 1-R77 must equal 0.
4. Riitters et al. (2000) developed a fragmentation classification based on (R81, R77).

- **R82. Ratio of the frequencies of two specific pixel values.**

Description. Ratio of the frequencies of two specified pixel values.
Notes:
1. Calculated as the number of pixels of the “first target code” divided by the number of pixels of the “second target code”. The order of target codes is important.
2. If byte output is requested, the calculated ratio is mapped to values in [1,255] as follows. Let n1 and n2 be the number of pixels of target codes 1 and 2 respectively.
If n1 > 0 and n2 = 0 then value = 255.
If n1 = 0 and n2 > 0 then value = 1.
If n1=n2=0 then value = 0 (missing).
If n1=n2 then value = 128.
If n1 > n2 the value increases with n1 from 129 to 254.
If n1 < n2 the value decreases with n2 from 127 to 2.

- **R83. Combined ratio of two specific pixel values.**

Description: Ratio of the frequency of a specified pixel value to the sum of frequencies of that pixel value and a second specified pixel value.
Notes:
1. Calculated as the number of pixels of "first target code" divided by the sum of the numbers of pixels of "first target code" and "second target code". The order of target codes is important.

5. Compiling and using spatcon outside GWB_SPATCON.

This section is relevant ONLY if you are NOT using GWB_SPATCON to access *spatcon*. If you ARE using GWB_SPATCON then ignore this section and refer to the GWB_SPATCON usage notes (Section 3).

Compiling *spatcon.c* produces a command-line executable.

Compile on Linux gcc:
```
gcc -std=c99 -m64 -O2 -Wall -fopenmp spatcon.c -o spatcon -lm
```

Compile on Windows with mingw64:
```
gcc -std=c99 -m64 -O2 -Wall -fopenmp Spatcon.c -o Spatcon -lm -D__USE_MINGW_ANSI_STDIO
```

Spatcon ran originally in "classic mode" which has some strange handling of I/O filenames. It is normally easier to run in "GUIDOS" mode which hardwires all the I/O filenames.

Spatcon recognizes an optional environment variable to specify number of cores to use. If this is not specified, then *spatcon* will use the maximum available in the shell.
Examples for N cores:
```
    Linux *csh family of shells: setenv OMP_NUM_THREADS N
    Linux all other shells: export OMP_NUM_THREADS=N
    Windows: set OMP_NUM_THREADS=N
```

Classic Mode:
See Section 3 for description and format of parameter file.
```
./spatcon <arg1> <arg2> <arg3>
        arg1 =input file (8-bit; note that a ".bsq" extension will be added by spatcon)
        arg2 =output file (note that a ".bsq" extension will be added)
        arg3 =parameter file
```
The program looks in the current directory for:
    Required: An 8-bit bsq input file "<arg1>.bsq".
    Required: A text file "<arg1>.siz" which contains two lines:
        nrows xxxxx   where xxxx is the number of rows in the input file
        ncols yyyyy   where yyyy is the number of columns in the input file
    Required: A text file (any name) which contains the run parameters
    Optional: If re-coding is requested, a text file "<arg1>.rec" which contains a re-coding lookup table.
Output. The program writes the output file "<arg2>.bsq" into the current directory.
Note: all filenames have a 500-character limit.

Guidos Mode:
See Section 3 for description and format of parameter file.

```
./spatcon
```

The program looks in the current directory for:

```
    Required: "scinput" an 8-bit input file
    Required: "scsize.txt" which contains two lines:
       nrows xxxxx   where xxxx is the number of rows in the input file
       ncols yyyyy   where yyyy is the number of columns in the input file
    Required: "scpars.txt" parameter file
    Optional: If recoding is requested, a text file "screcode.txt" a re-coding lookup table
```

Output. The program writes the output file "scoutput" into the current directory

Format of optional re-code file:
Important: The spatcon parameters m, a, and b refer to values AFTER optional re-coding.
Each line has two numbers separated by a space: <old value> <space> <new value>
Old values that are not listed will not be re-coded.
All old and new values must be in the range [0,255].
The old values can appear in any order.
If an old value appears on more than one line, the last one listed will be used.
Hints:
Can be used to set the m, a, and b parameters according to subsets (aggregations) of the input pixel values.
Example:

```
25 0 (if the m parameter is zero, then pixel value 25 is considered missing)
15 1
16 1 (In this case, both input values 15 and 16 are both re-coded to value = 1)
18 134
18 55 (In this case the old value 18 is new value 55, not 134)
...
```

File formats:
GWB_SPATCON accepts input as 8-bit TIF images and automatically converts *spatcon* output to a 8-bit or 32-bit TIF images. When used alone, *spatcon* I/O supports only the BSQ data format, which is sometimes referred to as ENVI format, which is a “flat, binary” series of 8-bit (or 32-bit) pixel values. Importantly, *spatcon* does not support the standard BSQ “header” files, and a little work is needed to set up the *spatcon* input “size” parameter file, and the metadata files that image format translators expect to convert *spatcon* BSQ output to some other format.

To prepare an input BSQ, one option is gdal_translate, for example:
gdal_translate -if GTiff -of ENVI -ot Byte <filename>.tif <spatcon input file name>.bsq”
Another option is QGIS, exporting an image in ENVI format with output filename ending in “.bsq”
Another GIS option is ArcMap, exporting an image in BSQ format.
The choice of output file names should conform to either ‘classic’ or ‘GUIDOS’ mode (see above).

To prepare the required “size” parameter file for *spatcon* (see above), the user must inspect the resulting *.hdr files to learn the number of rows and columns in the image. The number of rows is often labeled as “lines” or “NROWS”, and the number of columns is often labeled as “samples” or “NCOLS”.

To view the output file in an image viewer or use the file in a GIS, it is necessary to construct appropriate "header" files for the output BSQ image. When the *spatcon* output file is 8-bit, this can usually be done by copying the input header file to the output header file, e.g., from <input>.hdr to <output>.hdr (some GIS's also require copying the *.prj file, but don't copy any other files), and then running the appropriate gdal_translate or GIS import command.

When the *spatcon* output file is 32-bit, the same general procedure can be used, except the header file (*.hdr) must be edited after copying, so that the output file is interpreted as 32-bit values instead of 8-bit values. When using the ENVI specification, the "data type" should be 3 (signed 32-bit integer) instead of 1 (8-bit unsigned integer). With ArcMap BSQ's it's a little more complicated, changing "NBITS" from 8 to 32, and changing both "BANDROWBYTES" and "TOTALROWBYTES" to be 4 times the value given for the input 8-bit image, and ensuring the "NODATA" value (if used), is zero.

The above suggestions do not cover the range of format or software possibilities but should be enough to guide a knowledgeable user.

6. *Spatcon* history.

In 1992 my transition into the field of landscape ecology was marked by the development of a computer program (*landstat.c*; ***land****scape* ***stat****istics*[6]) to calculate a suite of "landscape pattern" metrics from a categorical raster map (Riitters et al. 1995). In 1994, inspired by Baker and Cai (1992), I developed a moving window implementation (*spatcon.c;* ***spat****ial* ***con****volution*) for several of the *landstat* metrics, and added a few more, thus permitting pixel-level mapping of landscape patterns as measures of landscape context (Riitters et al. 1997). *Spatcon* and *landstat* were personal research tools for exploring the questions of what to measure, how, and why (Riitters 2019). As such, documentation and distribution were always a low priority, and code obfuscation is the natural result of 30 years of modifying the original code only enough to not break it.

As part of a long-standing research collaboration, Peter Vogt (European Commission, Joint Research Centre, Ispra) began including a *spatcon* executable in some of his image analysis software (Vogt and Riitters 2017; Vogt et al. 2022), thus providing both desktop and server-based access to several of the *spatcon* metrics which had proven most useful in our collaborative research. Peter also developed user guides to explain what those metrics are and how they can be used within his GuidosToolBox and GuidosToolbox Workbench. In 2022, Peter suggested that some users may be interested in some of the original yet "hidden" *spatcon* metrics, which prompted this current effort to better document what those metrics are for the GWB_SPATCON module.

[6] Originally "land.c (air)," to measure sunfleck patterns on photographs, but never used for that purpose.

7. Programming notes

The *spatcon* author is not a professional programmer. As research tools, the code was typically obfuscated, the documentation did not exist, there was no tech support, and the error messages were enigmatic, yet it was all sufficient for my purposes. The objective of the current effort is to improve the documentation for use of *spatcon.c* within the GWB_SPATCON module.

The moving window algorithm in *spatcon* is very efficient because (a) the entire window needs to be evaluated only for the first pixel of a given row of the map – for subsequent steps in that row the accumulator is updated only according to the changing contents of the leading and trailing columns of the window[7], and (b) the algorithm is parallelized over the input rows[8].

The evolving uses of *spatcon* took advantage of improved hardware without making substantial changes to the code. The original coding was on a PC-386[9] which was okay for coding and testing, and while my work environment provided a Sparc10 (64MB RAM), there was still a problem with *spatcon* applications to the relatively large, high-resolution digital raster maps which were becoming available by the mid-90's (Riitters et al. 1997). This problem was solved by tiling the input map into overlapping tiles (*splitter.c*), executing *spatcon* on each tile, and collecting the output tiles into a single map at the end (*lumper.c*). In this way it was feasible to process the first available 30-m resolution land cover map of the continental U.S. (~9.1 x 10^9 pixels) on a new Ultra60 (1.5GB RAM) as if it was just one big map (Riitters et al. 2002). In 2002 I began automating tile processing on a Linux cluster[10] with the Portable Batch System. The cluster became obsolete with the acquisition of a dual quad-core workstation with 128GB RAM in 2012; GOMP-assisted parallel processing[11] came soon after, as the major hardware constraint shifted from "RAM-limited" to "core-limited." With parallel processing on modern (2022) workstations, disk I/O typically accounts for >95% of the total execution time.

Compared to the original code in 1994, most code changes involved simplifications rather than improvements. There is no longer a need to support, among other things, byte-swapping (Motorola/Intel chips), conditional compilation (ANSI/K&R C), weird data formats (e.g., *.pcx, *.xwd, and Arc/INFO ASCII), or user selection of the window step size and output pixel resolution. The important improvements include the addition of several metrics and parallel processing, and compatibility with Peter Vogt's GuidosToolBox and GuidosToolbox Workbench.

[7] This "subtract from the left and add from the right" logic was self-evident when working on a 16 Mhz 386 chip in 1994, but someone did publish a paper on it ~25 years later. I also did not publish my independent rediscovery of two other famous algorithms while writing *landstat.c* – flood-filling for patch identification and wall-following for perimeter identification.

[8] It cannot be parallelized over the columns in a row because the accumulator is updated serially within a row.

[9] Around 1990 I recycled some 286 components to build a 386 machine, with appreciation for Aubrey Pilgrim's 1988 book "Build your own 80386 IBM compatible and save a bundle." I learned C on DOS via Watcom C 8.0/386+DOS4G extender to take advantage of the 386's 32-bit memory addressing.

[10] A compute node was typically a rack-mounted dual Pentium III 866, 1.5GB RAM, small disk, and network card; all cheap and unreliable from eBay. The Ultra60 became the head node serving a NFS filesystem and running shell scripts to automate batch queue submissions. This homemade cluster achieved a maximum of 24 cores running on 14 nodes.

[11] Remarkably, only one line of new code and rearrangement of accumulator memory management.