Modern Spatial Econometrics in Practice: A Guide to `GeoDa`, `GeoDaSpace` and `PySAL`

Luc Anselin Sergio J. Rey

November 10, 2014

Chapter 3

Spatial Weights: Contiguity

Spatial weights are a key component in any cross-sectional analysis of spatial dependence. They are an essential element in the specification of the spatial variables in a model, such as the spatially lagged dependent variable and spatially lagged explanatory variables, as shown in Equations 1.1 and 1.2 for the spatial lag and spatial error model.

Formally, the weights express the neighbor structure between the observations as a $n \times n$ matrix $\mathbf{W}$ in which the elements w_{ij} of the matrix are the spatial weights:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{bmatrix}.$$

The spatial weights w_{ij} are non-zero when i and j are neighbors, and zero otherwise. By convention, the self-neighbor relation is excluded, so that the diagonal elements of $\mathbf{W}$ are zero, $w_{ii} = 0$.

In its most simple form, the spatial weights matrix expresses the existence of a neighbor relation in binary form, with weights 1 and 0. Formally, each spatial unit is represented in the matrix by a row i, and the potential neighbors by the columns j, with $j \neq i$. The existence of a neighbor relation between the spatial unit corresponding to row i and the one matching column j follows then as $w_{ij} = \mathbf{W}_{i,j} = 1$.

There are many criteria on which the construction of the spatial weights can be based. A comprehensive discussion is beyond the current scope. We focus on the two most common operational approaches and distinguish between a neighborhood relation based on the notion of contiguity and one derived from distance measures. Intrinsically, contiguity is most appropriate for geographic data expressed as polygons (so-called areal units), whereas distance is suited for point data, although in practice the distinction is not that absolute. In fact, polygon data can be represented by their centroid or central point, which then lends itself to the computation of distance. Similarly, a tesselation can be constructed for point data (e.g., Thiessen polygons), which allows for the determination of contiguity relationships between the polygons in the tesselation.

In this Chapter, we restrict our attention to contiguity weights. We first define some basic concepts and then proceed with a description of the weights functionality in each of `GeoDa`, `GeoDaSpace` and `PySAL`. Distance-based weights are covered in Chapter 4.

3.1 Basic Principles

3.1.1 Rook and Queen Contiguity

Contiguity means that two spatial units share a common border of non-zero length. Operationally, we can further distinguish between a *rook* and a *queen* criterion of contiguity, in analogy to the moves allowed for the such-named pieces on a chess board. The rook criterion defines neighbors by the existence of a common edge between two spatial units. The queen criterion is somewhat more encompassing and defines neighbors as spatial units sharing a common edge or a common vertex.[1] Therefore, the number of neighbors according to the queen criterion will always be at least as large as for the rook criterion.

In practice, the construction of the spatial weights from the geometry of the data cannot be done by visual inspection or manual calculation, except in the most trivial of situations. To assess whether two polygons are contiguous requires the use of explicit spatial data structures to deal with the location and arrangement of the polygons.

The way in which contiguity between polygons can be assessed in a GIS depends on its internal representation of the geometric features. The easiest situation is when the data representation in the GIS already captures topology, e.g., by using a node-arc-area approach or a connected edge list to store the information about the arrangement of the spatial features, as is the case in some of the spatial databases supported by `GeoDa`. However, in each of our three software implementations, the polygons are stored internally and the contiguity relations need to be constructed from an explicit matching of the boundary information for each pair of polygons. A brute force comparison of

[1]A third notion, referred to as *bishop* contiguity, is based on the existence of common vertices between two spatial units. It is seldom used in practice, and will not be considered here.

all pairs is highly inefficient and therefore refined geocomputational techniques are implemented.

It is important to keep in mind that the spatial weights are critically dependent on the quality of the GIS from which they are constructed. Problems with the topology in the GIS (e.g., slivers) will result in inaccuracies for the neighbor relations included in the spatial weights. In practice, it is essential to check the characteristics of the weights for any evidence of problems (see Section 3.1.5). When problems are detected, the solution is to go back to the GIS and fix or clean the *topology* of the data set. This is a routine operation in most GIS software.

3.1.2 Block Weights

A slightly different concept of neighbors follows when a block structure is imposed, in which all observations in the same block are considered to be neighbors. This is an example of a hierarchical spatial model, in which all units that share a common higher order level are "contiguous." For example, this applies readily to all counties in a state, census blocks in a census tract, and similar multilevel structures. This approach became more commonly known after its application in a study of innovation adoption by Case (1991, 1992). In the spatial econometric literature, it is sometimes referred to as *Case weights*.

The result is a block-diagonal spatial weights structure, in which all the spatial units in a block are neighbors, but there is no neighbor relation that spills over across blocks. In this respect, block weights are similar to the *regime* weights used in the treatment of spatial heterogeneity in Chapters 12 and 13.

3.1.3 Higher Order Contiguity

Up to this point, we have only considered the notion of a direct neighbor, or, more precisely, a *first order* neighbor. This concept can be generalized to allow for higher order neighbors, similar to the time series context, where a time shift can pertain to multiple periods. A higher order neighbor is defined in a *recursive* fashion, as a first order neighbor to a lower order neighbor. More formally, j is a neighbor of order k to i if:

- j is a first order neighbor to h,
- h and i are neighbors of order $k - 1$,
- j is not already a lower order neighbor to i.

Using this logic, candidates to be a second order neighbor would be any first order neighbor to another observation that is already a first order neighbor. However, this also has to be be limited to only those locations that are not already first order neighbors, to avoid duplication.

An efficient algorithm that construct higher order contiguity weights while removing redundant and circular paths is given in Anselin and Smirnov (1996). It is the approach implemented in all three software packages.

3.1.4 Transformation of Weights

In practice, the spatial weights are seldom used in their binary form, but subject to a transformation or standardization. We consider three common procedures: row-standardization, double standardization and variance stabilizing.

3.1.4.1 Row-Standardization

Row-standardization takes the given weights w_{ij} (e.g, the binary `0-1` weights) and divides them by the row sum:

$$w_{ij(s)} = w_{ij} / \sum_j w_{ij}.$$

As a result, each row sum of the row-standardized weights equals `1`. Also, the sum of all weights, $S_0 = \sum_i \sum_j w_{ij}$, equals n, the total number of observations.[2] *Row-standardization is the default approach in all three software packages.*

3.1.4.2 Double Standardization

Double standardization turns the weights matrix into a stochastic matrix, such that the sum of all the elements equals `1`:

$$w_{ij(ds)} = w_{ij} / \sum_i \sum_j w_{ij}.$$

As a result, for the new weights, $S_0 = 1$.

3.1.4.3 Variance Stabilizing

The variance stabilizing transformation was suggested by Tiefelsdorf et al. (1999) in the context of inference for Moran's I statistic in the

[2]Strictly speaking, this is only correct in the absence of so-called isolates, i.e., observations without neighbors (see Section 3.1.5.1). With q isolates, the sum $S_0 = n - q$.

presence of heteroskedastic data. It consists of a two-step transformation of the original weights w_{ij}. First, each element is divided by the square root of the row sum of squared weights:

$$w_{ij}^* = w_{ij} / \sqrt{\sum_j w_{ij}^2}$$

In the second step all the weights are rescaled by a factor n/Q, where $Q = \sum_i \sum_j w_{ij}^*$. This standardization is included for the sake of completeness, but is seldom used in practice.

3.1.5 Characteristics of Weights

The spatial weights matrix can be conceptualized as a mathematical expression for the structure of a network, where the spatial observations (locations) are nodes and the existence of a "neighbor" relation corresponds to a link. The characteristics of this network structure can be summarized by means of several statistics. In the three software packages, this is limited to some basic descriptive statistics derived from the number of neighbors for each location, the so-called neighbor cardinality. These include the minimum, maximum, mean and median number of neighbors, the number of non-zero weights, and the proportion of weights that are non-zero (an indication of the *sparseness* of the weights matrix).

A straightforward visualization of the structure of the weights matrix is obtained by means of a so-called *connectivity histogram.* This histogram shows the frequency of occurrence for each number of neighbors. In other words, each bar in the histogram gives how many spatial units have the corresponding number of neighbors. Ideally, the histogram is symmetric around a single mode, without any extremely high number of neighbors. While this is typically the case for contiguity weights, distance weights can result in a very large range of neighbor cardinality (see Chapter 4).

Two particular features of the connectivity distribution need to be taken into account. One is when the distribution is clearly bi-modal. This typically occurs when the spatial layout of observations is non-standard. For example, a situation where there are spatial units fully enclosed within another spatial unit (e.g., city counties in the U.S. state of Virginia), the connectivity histogram will have a mode at `1` (for the enclosed units, which all have exactly one neighbor, i.e., the enclosing unit), and a remainder distribution with a mode around `5` or `6`. The bimodal nature of the distribution will affect the interpretation of the spatially lagged variables (see Section 3.1.6).

A second feature is the occurrence of islands or isolates, to which we turn next.

3.1.5.1 Isolates

A particularly undesirable feature of the neighbor count distribution is when some locations do not have neighbors. In the associated spatial weights matrix, all elements in the row corresponding to such a location are zero: $w_{ij} = 0, \forall j$. Such observations are referred to as *isolates* or *islands*. They are easy to identify in the connectedness statistics, when the minimum number of neighbors is zero, or in the connectivity histogram, when a bar is present for the value zero.

The isolates may be proper, e.g., when there are true islands included in the data set, or they may be the result of problems in the creation of the partial weights (e.g., in the case of an incorrect topology for the data). In either case, the inclusion of isolates in a spatial econometric analysis causes complications. The main issue is whether or not to keep the observations in question as part of the data set. There are two aspects to this problem.

A first aspect pertains to the consequences in terms of the purely *spatial* characteristics of a data analysis. Since the corresponding weights are all zero, any averaging of neighboring locations will yield zero as the result. The zero value therefore essentially eliminates the isolate from any consideration of spatial effects.

In a spatial regression specification, the inclusion of a spatially lagged variable introduces zero values for the islands (see Section 3.1.6). This may bias the estimate for the autoregressive parameter, unless it is properly accounted for in the estimation algorithm.

The complications resulting from isolates in the spatial analysis would suggest that they should be eliminated from the data set. This has great intuitive appeal, since spatial analysis is about interaction and isolates do not interact.

A second aspect of the isolate problem pertains to the loss of degrees of freedom that would result from dropping unconnected observations from the analysis. For the non-spatial aspects of the analysis, such as the estimation of regular, non-spatial parameters in a model, this may affect the precision of the results. Note that in contrast to the *spatial* analysis, the non-spatial aspects of the econometric model are not affected by the presence of isolates. Therefore, non-spatial aspects of the analysis, such as the estimation of regular, non-spatial parameters in a model, may lose precision as a result of dropping the isolates from the data set. This is a particular concern when the number of observations is small. In large data sets ($n > 10000$), this is unlikely to have any practical effect, as long as the number of isolates is small.

In practice, one needs to make a careful decision and evaluate whether the loss of some degrees of freedom is outweighed by the greater purity of the spatial analysis. In any event, when the isolates are not dropped, it is important to ensure that the results of the analysis are interpreted correctly.

3.1.6 Spatially Lagged Variables

With a neighbor structure defined by the non-zero elements of the spatial weights matrix $\mathbf{W}$, a spatially lagged variable is a weighted sum or a weighted average of the neighboring values for that variable. In our notation, we designate the spatial lag of y as Wy. As a result, for observation i, the spatial lag of y_i, referred to as $[Wy]_i$ (the variable Wy observed for location i) is:

$$[Wy]_i = w_{i,1}y_1 + w_{i,2}y_2 + \cdots + w_{i,n}y_n,$$

or,

$$[Wy]_i = \sum_{j=1}^{n} w_{i,j}y_j,$$

where the weights $w_{i,j}$ consist of the elements of the i-th row of the matrix $\mathbf{W}$, matched up with the corresponding elements of the vector $\mathbf{y}$. In other words, this is a weighted sum of the values observed at neighboring locations, since the non-neighbors are not included (those i for which $w_{ij} = 0$). Typically, the weights matrix is very sparse, so that only a small number of neighbors contribute to the weighted sum. For row-standardized weights, with $\sum_j w_{ij} = 1$, the spatially lagged variable becomes a weighted average of the values at neighboring observations.

In matrix notation, the spatial lag expression corresponds to the matrix product of the $n \times n$ spatial weights matrix $\mathbf{W}$ with the $n \times 1$ vector of observations $\mathbf{y}$, or $\mathbf{W}.\mathbf{y}$. The matrix $\mathbf{W}$ can therefore be considered to be the spatial lag *operator* on the vector $\mathbf{y}$.

It is seldom necessary to create a spatially lagged variable explicitly in any of the three software packages. For all the spatial models, this is done internally by the software. For example, in a spatial lag model, the spatially lagged dependent variable Wy is computed directly without user intervention from the weights information provided in the model specification. Similarly, in the estimation of the spatial lag model by means of IV/GMM (Chapter 7), the spatially lagged explanatory variables $\mathbf{WX}$ that are used as instrumental variables are computed under the hood and do not need to be specified explicitly. There are however some instances in which selected spatially lagged explanatory variables may be included on the right hand side of the equation, as in a so-called spatial cross-regressive specification (Florax and Folmer 1992). In those instances, the spatially lagged variables need to be created and included explicitly into the model specification, since the software does not distinguish between a regular and a spatially lagged explanatory variable.

3.1.7 Weights File Formats

In practice, contiguity-based weights are typical extremely sparse, meaning that the weights matrix mostly consists of zero elements. For example, using rook contiguity for the continental U.S. counties ($n = 3085$) yields an average of `5.6` neighbors for each county. This corresponds to only `17,188` out of the `9,514,140` off-diagonal elements of the $n \times n$ weights matrix being non-zero, or `0.18`%. Hence, in actual computations, it is important to store the weights efficiently in a sparse format and thereby avoid allocating storage space for the many zeros. Several formats have been suggested to accomplish this in practice.

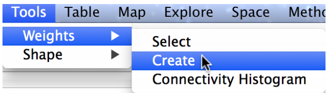
Arguably the most commonly used file format for sparse contiguity information is the so-called `GAL` format, introduced in the `SpaceStat` software package in 1995, and since adopted by many others, including `GeoDa`, `STARS` and the `spdep` library in `R`.[3] The `GAL` format consists of two parts to store the contiguity information for each observation. In a first part (typically entered on a separate line), an identifier for the observation is given, followed by the number of neighbors. Next follows a line containing the identifiers for the neighbors. The identifier in question must be unique, to properly match the attribute information for the observation to its contiguity structure.

In `GeoDa`, the `GAL` weights file also contains a header line (i.e., the first line in the file), with metadata such as the number of observations, the name of the shape file from which the contiguity was derived, and the variable name (field in the data base) for the identifier. `GeoDaSpace` and `PySAL` also support a `GAL` format with only the number of observations in the header line. Note that the `GAL` file only stores the presence of contiguity, but not a value for the spatial weights.

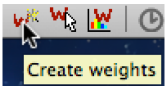
A second major file format for sparse weights is the so-called `GWT` format, also initially introduced in `SpaceStat`. For each pair of neighboring observations, a record consists of the triplet i, j, w_{ij}, where i is the `ID` for the *origin* spatial unit (i.e., the unit under consideration), j is the `ID` for the *destination* unit (i.e., the neighbor), and w_{ij} is the value for the weights. For contiguity weights, the latter is always `1`. `GWT` weights are more appropriate for the storage of distance-based weights, and will be revisited in Chapter 4.

All three software packages support saving and reading spatial weights in `GAL` and `GWT` formats. In addition, `GeoDaSpace` and `PySAL` can also read and write a number of alternative weights formats.

[3]The origins of the GAL format are the proposals formulated in the Geographical Algorithms Library in the United Kingdom in the late 1980s.



(a) Menu



(b) Toolbar

Figure 3.1: GeoDa weights creation

3.2 Contiguity Weights in GeoDa

In GeoDa, spatial weights can be constructed in a project from any file, data base or web feature service that was specified as the input source (see Section 2.1). Once a project is active, the current layer is used as the input and need not be specified explicitly (in contrast to what held for versions of GeoDa prior to 1.6). Weights creation is invoked either from the menu, using `Tools > Weights > Create` (see Figure 3.1a), or using the left-most icon in the weights toolbar (Figure 3.1b). An existing file can be read by clicking on the middle icon of the weights toolbar, labeled `Open weights file`, or by using `Tools > Weights > Select` from the menu.

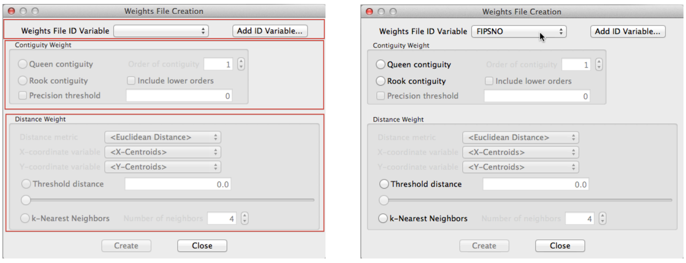
In addition, in the context of a regression specification, an existing weights file can be read by selecting its file name, or a new weights file can be created by clicking on the corresponding icon in the regression specification dialog (Figure 2.10). For example, to specify the spatial weights to carry out diagnostics for spatial autocorrelation in the context of an OLS regression, the weights need to be selected in the dialogs shown in Figures 5.9 and 5.10. This is discussed more fully in Section 5.2.4 of Chapter 5. The same process is required for the maximum likelihood estimation of the spatial lag and spatial error models in GeoDa (see Chapters 8 and 10).

In the spatial regression functionality of GeoDa, the spatial weights are always used in *row-standardized* form.

3.2.1 The Weights Interface

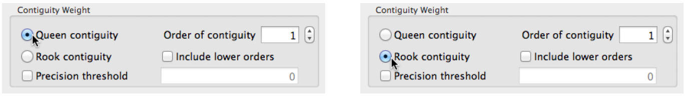
GeoDa's weights file creation dialog is shown in Figure 3.2a. It consists of three main parts, delineated by the red boxes in the Figure. The top box contains the specification of the `ID Variable`. This is a variable that contains a unique integer value for each observation, such that the rows in the spatial weights matrix can be accurately matched with the observations in the data set. For example, in Figure 3.2b, the `ID Variable` has been set to `FIPSNO`.

When no acceptable `ID Variable` is contained in the data set, one can be included by clicking on the `Add ID Variable` button. A common issue encountered in practice is that a variable that seems to take on integer values is actually stored as real in the data base. As a



(a) GeoDa weights dialog (b) ID variable specified

Figure 3.2: GeoDa weights creation dialog



(a) Queen contiguity (b) Rook contiguity

Figure 3.3: Queen and Rook contiguity in GeoDa

consequence, such a variable would not be recognized as a proper ID Variable. The Add ID Variable function will ensure that an integer sequence number is inserted in the data set. This variable will either have the default label POLY_ID, or any other unique variable name specified in the dialog by the user.

The second box in the weights file creation dialog pertains to contiguity weights, and is further discussed next. The third box deals with the construction of distance based weights, covered in Chapter 4.

3.2.2 Creating Contiguity Weights

3.2.2.1 First Order Contiguity

Contiguity weights using either the queen criterion or the rook criterion are created by checking the corresponding radio button in the dialog, as shown in in Figures 3.3a and 3.3b. The default is first order contiguity, evidenced by the presence of the value 1 in the Order of contiguity box.

At this point, clicking the Create button will bring up the customary File Save dialog and request a name for the new weights file. In the example shown in Figure 3.4, we used NAT_test.gal as the file name. In fact, the queen weights are the same as those contained in the sample data file nat_queen.gal, but we use a different file name here in order not to overwrite the sample file.

The contents of the GAL format file for the first three observations are illustrated in Figure 3.5. The first line is a header line listing 0, a

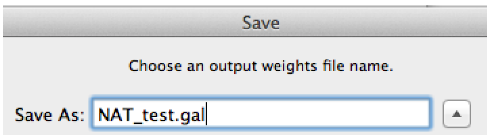


Figure 3.4: Spatial weights save file dialog

```
0 3085 NAT FIPSNO
27077 3
27007 27135 27071
53019 3
53047 53065 53043
53065 4
53043 53051 53063 53019
```

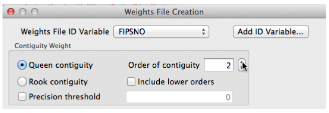
Figure 3.5: GAL weights file format

placeholder for future functionality, 3085, the number of observations, NAT, the name of the source shape file, and FIPSNO, the ID Variable. Next follow three sets of two lines, one for each observation. The first set has as first line the ID of the observation, 27077, followed by the number of neighbors, 3. Next follow the ID values for those neighbors: 27007, 27135, and 27071. The same sequence is used for the other observations. The GAL file is a simple text file, and thus can be easily edited. However, extreme care should be used when attempting this, especially to ensure that symmetry is maintained.

In addition, the Contiguity Weight dialog also contains a check box for Precision threshold. In most instances, this box should be kept *unchecked*. It should only be used when there are small inaccuracies present in the GIS and the user has a good sense of the order of magnitude of these inaccuracies. With a Precision threshold other than 0, a fuzzy comparison is carried out to determine whether two polygons have a vertex in common. The default is an exact comparison, which assumes that the topology of the underlying polygon is correct. The Precision threshold option provides a way to deal with small inaccuracies that avoids the need to go back to a GIS to fix the topology, but it is by no means a replacement for this GIS functionality.

3.2.2.2 Higher Order Contiguity

Higher order contiguity is obtained by changing the value for Order of contiguity from the default of 1 to any other value. For example, this is illustrated for second order contiguity in Figure 3.6a. An option for higher order contiguity is to include the lower order contiguity neighbors in the spatial weights. The default is not to include them

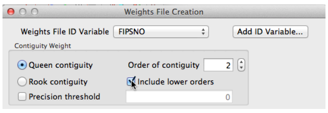

Weights File Creation

Weights File ID Variable: FIPSNO

Contiguity Weight

- [✓ Queen contiguity] — Order of contiguity: 2
- Rook contiguity — Include lower orders: unchecked
- Precision threshold: unchecked, 0

(a) Second order contiguity


Weights File Creation

Weights File ID Variable: FIPSNO

Contiguity Weight

- [✓ Queen contiguity] — Order of contiguity: 2
- Rook contiguity — [✓ Include lower orders]
- Precision threshold: unchecked, 0

(b) Second order inclusive

Figure 3.6: Higher order contiguity in GeoDa

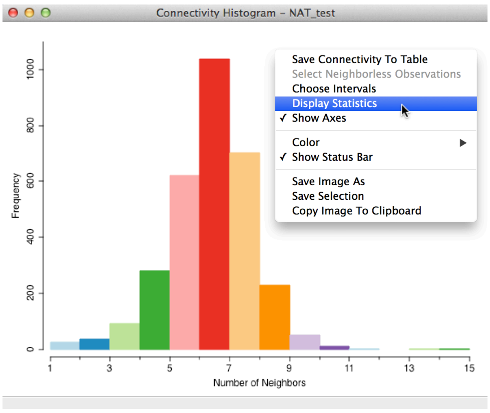


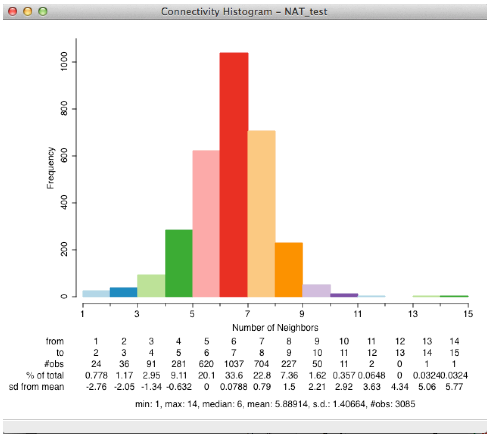
Figure 3.7: GeoDa connectivity histogram

(as in Figure 3.6a), so that the resulting weights have no redundant or circular paths. However, in some instances, it may be desirable to include lower order weights. For example, this would be a way to create a weights matrix that combines first and second order neighbors, in a case where the first order weights under-bound the range of interaction and there is no reason to introduce an additional parameter. In those instances, the box labeled `Include lower orders` should be checked, as in Figure 3.6b.

3.2.3 Weights Characteristics

In GeoDa, the weights characteristics are given as a `Connectivity Histogram`. This is invoked from the menu as `Tools > Weights > Connectivity Histogram` (see Figure 3.1a), or by clicking on the right-most icon in the weights toolbar (see Figure 3.1b). The result is a special GeoDa histogram, in which each bar shows how many spatial units have the number of neighbors shown on the horizontal axis, as in Figure 3.7.

This visual representation of the distribution of neighbor cardinalities can be further quantified by invoking the `Display Statistics` option of the histogram. This is carried out by right clicking on the



from	1	2	3	4	5	6	7	8	9	10	11	12	13	14
to	2	3	4	5	6	7	8	9	10	11	12	13	14	15
#obs	24	36	91	281	620	1037	704	227	50	11	2	0	1	1
% of total	0.778	1.17	2.95	9.11	20.1	33.6	22.8	7.36	1.62	0.357	0.0648	0	0.0324	0.0324
sd from mean	-2.76	-2.05	-1.34	-0.632	0	0.0788	0.79	1.5	2.21	2.92	3.63	4.34	5.06	5.77

min: 1, max: 14, median: 6, mean: 5.88914, s.d.: 1.40664, #obs: 3085

Figure 3.8: `GeoDa` connectivity histogram with statistics

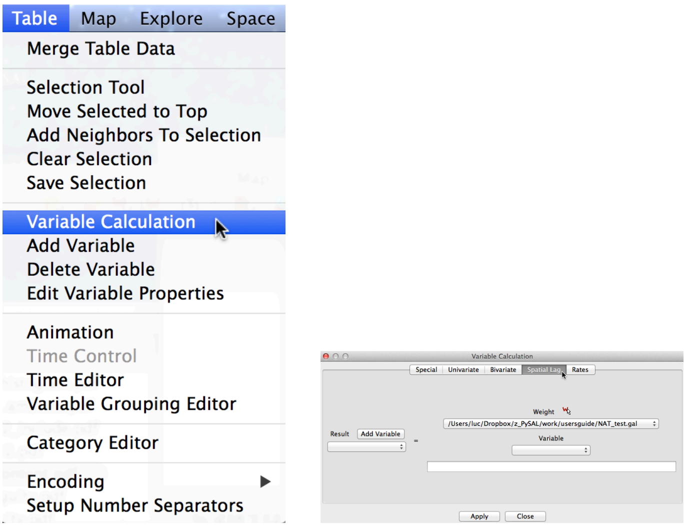
displayed window, as shown in Figure 3.7. The result is a collection of descriptive statistics listed below the graph, as in Figure 3.8. This is the standard set of statistics that accompanies the histogram functionality in `GeoDa`. Of particular interest in the current context is the bottom line, which lists the `min`, `max`, `median`, and `mean` number of neighbors, among others. In our example, the respective values are `1`, `14`, `6`, and `5.88914`.

In case there are isolates, not only will the `min` be `0` and there will be a bar corresponding to this value, but a warning message will be generated as well.

3.2.4 Constructing Spatially Lagged Variables

In `GeoDa`, the computation of spatially lagged variables is handled as part of the `Variable Calculation` functionality in the `Table` menu. From the main menu, it is invoked as shown in Figure 3.9a, by selecting `Table > Variable Calculation`. Alternatively, it can be started by right-clicking in the `Table` itself.[4] This brings up the `Variable Calculation` dialog, shown in Figure 3.9b. The `Spatial Lag` computation is selected by clicking on the fourth tab at the top of the dialog. If a spatial weights file has already been created, it will be listed in the dialog, as in our example, with the file `NAT_test.gal` listed under the label `Weight` in Figure 3.9a. If not, the weights creation icon needs to be invoked to create a weights matrix. The next steps consist of specifying a name for the new spatially lagged variable (shown in Figure 3.10a as `W_HR60`) and selecting the variable for which the lag

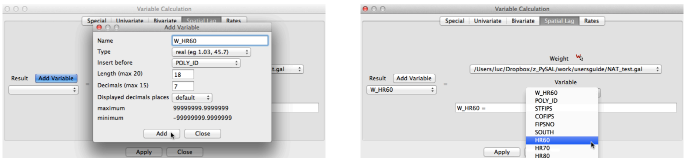
[4]The table is invoked by means of the fourth icon on the `GeoDa` toolbar (see Figure 2.1).



Result: Add Variable =
Weight: /Users/luc/Dropbox/z_PySAL/work/usersguide/NAT_test.gal
Variable

(a) Variable calculation (b) Spatial lag option

Figure 3.9: Spatial lag computation in `GeoDa` Table



Add Variable

Name	W_HR60
Type	real (eg 1.03, 45.7)
Insert before	POLY_ID
Length (max 20)	18
Decimals (max 15)	7
Displayed decimals places	default
maximum	99999999.9999999
minimum	-99999999.9999999

Result: W_HR60 =
Weight: /Users/luc/Dropbox/z_PySAL/work/usersguide/NAT_test.gal
Variable: W_HR60, POLY_ID, STFIPS, COFIPS, FIPSNO, SOUTH, HR60, HR70, HR80
W_HR60 =

(a) New variable name (b) Variable selection

Figure 3.10: Spatial lag computation variable selection in `GeoDa` Table

needs to be computed from a drop down list (shown in Figure 3.11 as `HR60`). Clicking `Apply` will compute the new variable and insert it into the data table, as illustrated in Figure 3.11. At this point, the variable can be used for any analysis in `GeoDa` and also included in a regression specification.

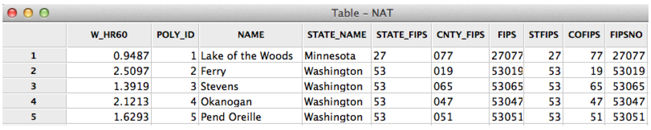


Table – NAT

	W_HR60	POLY_ID	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	COFIPS	FIPSNO
1	0.9487	1	Lake of the Woods	Minnesota	27	077	27077	27	77	27077
2	2.5097	2	Ferry	Washington	53	019	53019	53	19	53019
3	1.3919	3	Stevens	Washington	53	065	53065	53	65	53065
4	2.1213	4	Okanogan	Washington	53	047	53047	53	47	53047
5	1.6293	5	Pend Oreille	Washington	53	051	53051	53	51	53051

Figure 3.11: Spatial lag added to `GeoDa` Table

3.3 Contiguity Weights in GeoDaSpace

3.3.1 The Weights Interface

In GeoDaSpace, the functionality to construct and read spatial weights is contained in the Model Weights and Kernel Weights sections of the regression interface (see Figure 2.12). Contiguity weights are managed through the Model Weights interface, detailed in Figure 3.12a. Kernel Weights are a form of distance-based weights and are further discussed in Chapter 4.

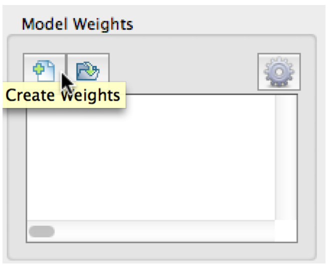
The functionality is invoked by means of the three icons on top of the dialog. The left-most icon is to Create Weights, the next icon to Open Weights from an existing spatial weights file, and the right-most (gear) button is to report the Properties of the weights. We cover each in turn.

3.3.2 Creating Model Weights

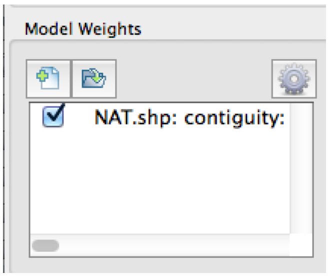
Selecting the Create Weights icon in the Model Weights panel brings up a dialog that is very similar to the layout used for GeoDa. Here too, the dialog lists the current shape file as the Input File and gives an option between Contiguity weights and Distance weights. We focus on the left tab for contiguity weights.

The Contiguity Weights dialog provides the choice between a radio button for queen and rook contiguity, as illustrated in Figure 3.13a for queen contiguity and Figure 3.13b for rook contiguity. Again, an ID Variable must be specified. In our example, we have selected FIPSNO. If no suitable ID Variable is available in the data set, clicking on the + sign will add a sequence number as the ID (the default variable name is POLY_ID).

The default for the contiguity weights is first order contiguity. Higher order contiguity weights can be constructed by specifying an

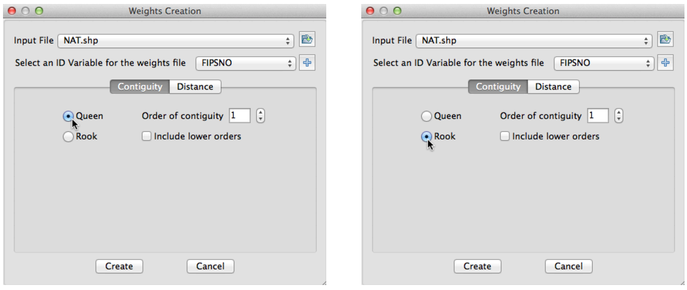


(a) Initial interface



(b) Populated interface

Figure 3.12: Model Weights interface in GeoDaSpace



Input File: NAT.shp
Select an ID Variable for the weights file: FIPSNO
[✓ Queen] Rook — Order of contiguity: 1 — Include lower orders: unchecked
Queen [✓ Rook] — Order of contiguity: 1 — Include lower orders: unchecked

(a) Queen contiguity (b) Rook contiguity

Figure 3.13: Contiguity weights in `GeoDaSpace`

`Order of contiguity` greater than 1. As in `GeoDa`, there is the option to include the lower order neighbors by marking the corresponding check box.

After clicking on the `Create` button, a file save dialog opens to specify a file name for the spatial weights. The file will be saved with a `GAL` file extension. In addition, a note will be placed in the `Model Weights` window indicating that a `contiguity` weights file has been created, as illustrated in Figure 3.12b. Note that this is not the file name (as in the example for `GeoDa`, we used `NAT_test.shp`), but gives the name of the original shape file (`NAT.shp`) followed by `contiguity`. This distinguishes the case where the weights are created on the fly from when they are read from a file (Section 3.3.3).

3.3.3 Reading Contiguity Weights

`GeoDaSpace` supports all the weights formats of the `PySAL weights` module. This includes not only the standard `GAL` and `GWT` formats, but also formats adopted by ArcGIS, MatLab, Stata, etc. The full list of supported formats can be seen from the file dialog that opens when the `Open Weights` menu icon is selected in the `Model Weights` panel of the `GeoDaSpace` GUI. The list of files that are enabled for opening is illustrated in Figure 3.14.

In order to read a weights file, select the file name in the open file dialog. When this is completed, the file name will be listed in the `Model Weights` panel. In our example, we used the sample file `nat_queen.gal` for queen contiguity among the continental U.S. counties. As a result, this filename is now listed in the panel, as shown in Figure 3.15. Note that in contrast with the previous Section, the actual name of the file is listed.

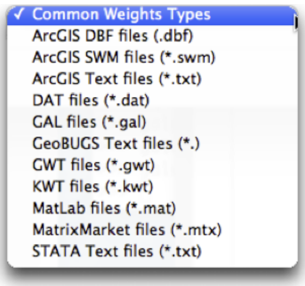


Figure 3.14: Weights formats in open file dialog

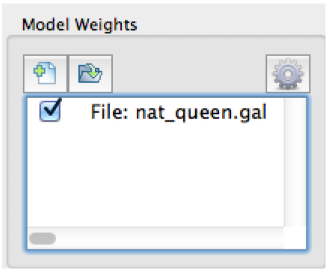
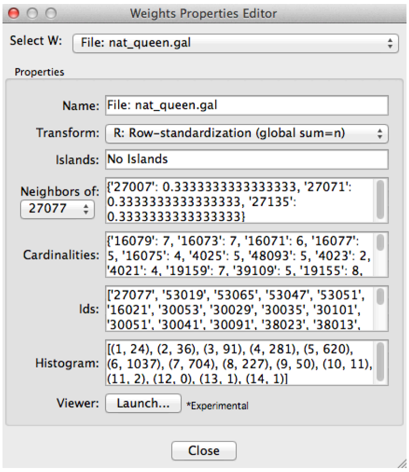


Figure 3.15: Queen contiguity weights file from `nat_queen.gal`

3.3.4 Weights Properties

The properties of the spatial weights can be accessed by selecting the gear icon in the `Model Weights` panel. This opens a panel for the



Select W:	File: nat_queen.gal
Name:	File: nat_queen.gal
Transform:	R: Row-standardization (global sum=n)
Islands:	No Islands
Neighbors of: 27077	{'27007': 0.3333333333333333, '27071': 0.3333333333333333, '27135': 0.3333333333333333}
Cardinalities:	{'16079': 7, '16073': 7, '16071': 6, '16077': 5, '16075': 4, '4025': 5, '48093': 5, '4023': 2, '4021': 4, '19159': 7, '39109': 5, '19155': 8,
Ids:	['27077', '53019', '53065', '53047', '53051', '16021', '30053', '30029', '30035', '30101', '30051', '30041', '30091', '38023', '38013',
Histogram:	[(1, 24), (2, 36), (3, 91), (4, 281), (5, 620), (6, 1037), (7, 704), (8, 227), (9, 50), (10, 11), (11, 2), (12, 0), (13, 1), (14, 1)]
Viewer:	Launch...

Figure 3.16: Properties of the `nat_queen.gal` weights file

`Weights Properties Editor` that lists several useful characteristics of the spatial weights, as shown in Figure 3.16.

At the top is a selection bar that lists the current weights file (`nat_queen.gal`). If other weights were created during the current session, they will also be accessible from this drop down list. The name of the weights is repeated in the box associated with the `Name` label, in our example `File: nat_queen.gal`, identical to the entry in the `Model Weights` panel.

Skipping to the fourth panel, labeled `Neighbors of`, we see the list of neighbors for observation with `ID` value `27077` (i.e., `FIPSNO = 27077`) as well as the values for the matching the weights. The neighbors are the counties with `FIPSNO` `27007`, `27071` and `27135`, with `0.333` as the value for the weights. Note how the contiguity information is identical to lines 3–4 in the `GAL` file shown in Figure 3.5. A similar listing for other observations is generated by scrolling down the list under the `Neighbors` label and selecting the desired `ID` value.

Finally, a complete list of values for the `ID Variable` is given under the `Ids:` label. It is alway useful to check this to ensure that no mismatches happen between the weights file and the data set.

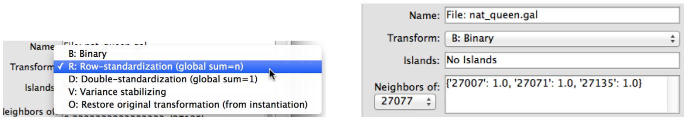
3.3.4.1 Weights Transformations

The second label in the `Weights Properties Editor` gives the current `Transform`. The default in `GeoDaSpace` is to treat all `Model Weights` in *row-standardized* form. This is indicated in the box as `R: Row-standardization (global sum = n)`. However, this option not only lists the current state of the weights object, but also allows one to change the transformation. In all, five different states are possible, as listed in Figure 3.17a:

- Binary, `B`
- Row-standardization, `R`
- Double-standardization (i.e., a stochastic weights matrix), `D`
- Variance stabilizing, `V`

as well as restoring the original state (before any transformation, as `O`).

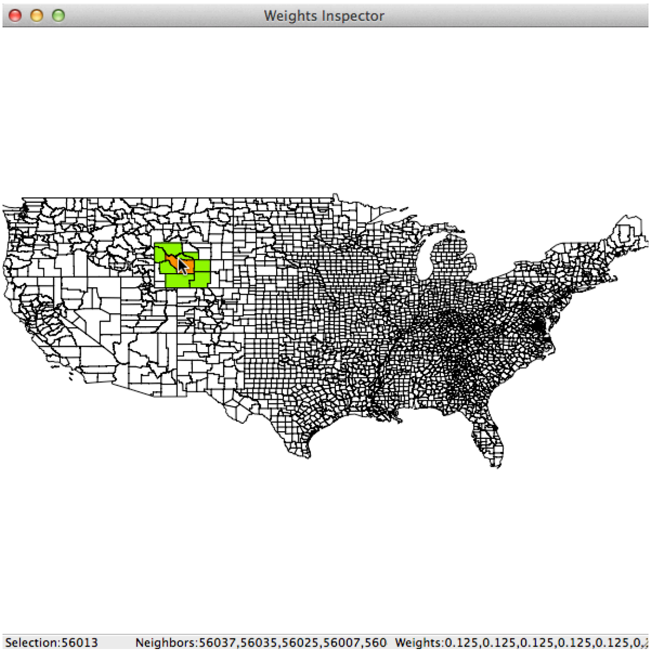
To assess the effect of a transformation, we use the drop-down list to select `B: Binary`. As a result, the value for the weights in the `Neighbors of` panel changes from `0.333` to `1.0`, as illustrated in Figure 3.17b. As stated before, all estimation procedures in `GeoDaSpace` require the spatial weights to be row-standardized, so this functionality should only be invoked by power users who are fully aware of the consequences of doing so.


Name
Transform
Islands
leighbors of
B: Binary
✓ R: Row-standardization (global sum=n)
D: Double-standardization (global sum=1)
V: Variance stabilizing
O: Restore original transformation (from instantiation)

Name: File: nat_queen.gal
Transform: B: Binary
Islands: No Islands
Neighbors of: 27077 {'27007': 1.0, '27071': 1.0, '27135': 1.0}

(a) Default (b) Binary

Figure 3.17: Weights transformations in GeoDaSpace



Selection:56013 Neighbors:56037,56035,56025,56007,560 Weights:0.125,0.125,0.125,0.125,0.125,0,

Figure 3.18: Weights connectivity viewer

3.3.4.2 Visualizing Weights Properties

The Weights Properties Editor contains three more panels with useful information about the characteristics of the spatial weights. The third panel in the Editor, labeled Islands indicates whether or not the weights yield unconnected observations. In our example, this is not the case, so that the box lists No Islands. If there had been isolates, the panel would contain the values for the ID Variable for those observations.

The Cardinalities are shown in the fifth panel. For each observation, listed by its ID, this gives the number of neighbors. This forms the basis for the construction of the Histogram. listed as the last panel. This is a tabular counterpart to the Connectivity Histogram

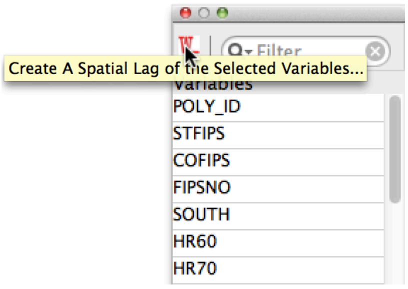


Figure 3.19: Creating a spatial lag from the variable list

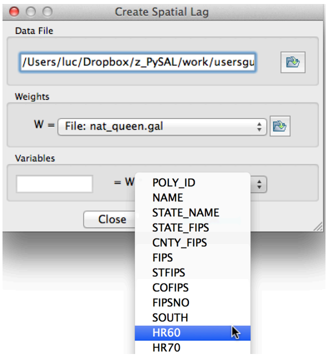
of Figure 3.7. Each pair lists the number of neighbors and how many observations have that many neighbors. In our example, we see that the minimum and maximum number of neighbors are `1` and `14`, with respectively `24` and `1` observations that match this cardinality.

A final feature of the `Weights Properties Editor` is an experimental contiguity `Viewer`. Clicking on the `Launch` button at the bottom of the `Editor` window starts an interactive viewer thats visualizes the neighbors of a selected unit based on the specified spatial weights. A file dialog will request the name for a matching shape file, which will then generate a map. One can use the mouse to brush over the map, such that for each selected spatial unit the corresponding neighbors (following the definition from the weights file) are highlighted on the map in the `Weights Inspector`, as illustrated in Figure 3.18 for the `NAT.shp` example shape file. The bottom of the window lists the `ID` of the selected unit as well as the `ID`s of the neighbors and the corresponding weights values. This is adjusted dynamically as the selection changes.

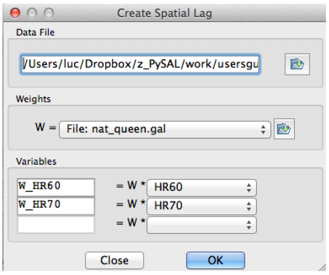
3.3.5 Creating Spatially Lagged Variables

A spatially lagged variable is created from the variable list in the `GeoDaSpace` GUI. As shown in Figure 3.19, this is invoked by clicking on the `W` icon at the top left of the variable list window.

Clicking the icon brings up a variable and weights selection dialog, as illustrated in Figure 3.20. An existing weights file is required, e.g., `nat_queen.gal` in the example shown, and the variables are selected from a drop down list, as shown in Figure 3.20a. Each time a variable is selected, a new variable name with a prefix `W_` is created and listed in the window, as in Figure 3.20b. Pressing the `OK` button brings up a file save dialog in which a name for a new data file (with file extension `dbf`) must be specified. In other words, the new spatially lagged variables are not added to the current data set, but included in a new `dbf` file



(a) Variable selection



(b) Variable names

Figure 3.20: Spatial lag dialog in GeoDaSpace

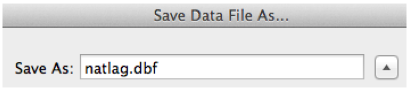


Figure 3.21: Saving the file with spatially lagged variables

(e.g., `natlag.dbf` in our example in Figure 3.21).

Upon completion of the file save process, the new data file is included as the data set in the GUI, with the spatially lagged variables added to the variable list, as shown in Figure 3.22. At this point, the variables are available to be included in any model specification.

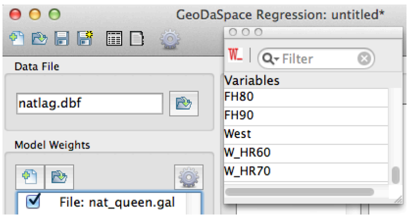


Figure 3.22: Spatially lagged variables in variable list

3.4 Contiguity Weights in PySAL

The spatial weights functionality of GeoDaSpace wraps a series of PySAL classes and methods and makes them available through a GUI. In the command line environment of the PySAL weights module, these methods can be invoked directly, with access to the full range of arguments and options. By contrast, even though GeoDaSpace uses the same code base, some of the options are limited by design to the most commonly used ones.

The estimation routines in spreg take advantage of the sparse matrix format supported by the Python scipy module (scipy is a stated dependency for PySAL). In GeoDaSpace, this is implemented in a transparent fashion, without any user intervention. In PySAL, two different spatial weights data structures are supported as Python classes. One, the weights object class W, is the most general spatial weights class and relies heavily on the use of dictionaries to store the observation IDs and associated weights. The other data structure implements the spatial weights as a scipy sparse array in the form of the WSP class.

The user regression functions in spreg take a spatial weights object as an argument and convert it internally to a sparse array format. For completeness sake, we briefly discuss the difference between the two data structures, but for all practical purposes, creating and reading/writing weights objects relies on the general weights class W.

In the remainder of this section, we review some of the most common contiguity weights operations contained in PySAL. This functionality is made available through so-called convenience or user classes, which can be called directly with a simplified namespace. The user classes provide a more user-friendly interface to a collection of underlying classes, methods and functions that carry out the actual operations. With a few exceptions, we limit the discussion to the functionality that matches GeoDaSpace, which is only a subset of the full weights operations available in PySAL.

For a complete listing of all options and arguments, we refer to the PySAL online documentation (tutorials and API Reference), and of course the source code itself.

We begin with a discussion of the two data structures for a spatial weights object. As a preliminary, we make sure that the numpy and pysal modules are imported:

```
>>> import numpy as np
>>> import pysal
```

3.4.1 Spatial Weights Object

3.4.1.1 The Spatial Weights Class W

The fundamental spatial weights class in PySAL is almost never created directly by the user, but is invoked by several functions that create weights from a shape file, or read the contents from an existing GAL file. However, it is important to understand the range of attributes associated with a spatial weights object, once created.

The only mandatory argument to create a spatial weights object is a Python *dictionary* labeled `neighbors` that contains a collection of *key-value pairs*, with the observation ID as the key, and a list with the IDs of the associated neighbors as the value. For example, for FIPSNO = 27077 in our U.S. county queen contiguity file (e.g., as listed in Figure 3.5), the corresponding entry would be (with the ID as an integer value):

```
27077 : [ 27007, 27135, 27071 ]
```

To make the discussion more concrete, we consider a simple example of a regular 3×3 grid with the 9 observations labeled $0, 1, 2, \ldots, 8$, starting in the upper-left corner of the grid and moving row by row. The first row thus consists of observations $0, 1, 2$, etc. We add a slight complication (the purpose of which will soon become clear) in the form of a 10-th observation (with label 9) that is unconnected to the others (i.e., an isolate). We specify the `neighbors` dictionary as follows:

```
>>> neighbors = {0: [3, 1], 1: [0, 4, 2], 2: [1, 5],
                 3: [0, 6, 4], 4: [1, 3, 7, 5], 5: [2, 4, 8],
                 6: [3, 7], 7: [4, 6, 8], 8: [5, 7],
                 9 : [ ]}
```

We use the rook criterion of contiguity so that the four corner cells each have two neighbors, the other edge cells (not corners) have three, and the center cell (labeled 4) has four neighbors. Cell 9 has an empty list for its neighbor value (unconnected).

The complete call to create a spatial weights object is:

```
>>> w = pysal.W(neighbors,weights=None,id_order=None,
                silent_island_warning=False)
```

Of the four arguments, only the `neighbors` dictionary is required. The optional arguments are:

- `weights`: a dictionary with as key the observation ID and as value a list of the weights for the neighbors; the weights must be given in the same order as the neighbors are listed in the matching `neighbors` list
- `id_order`: a list with the order in which the ID variables appear in the data array

- silent_island_warning: a boolean to indicate if the warning message for isolates needs to be turned off; the default is False, i.e., a warning is always generated

Typically, the values for the weights are equal for all the neighbors, so that the ordering requirement is less onerous than it may sound. When no weights are specified, the default value is 1.0. The id_order is important to make sure that the weights and the observations are aligned properly. This is necessary because a Python dictionary data structure is inherently unordered. The default is to use the order that results from a lexicographic sorting of the keys, but in some applications this may lead to unexpected results. It is always safest to specify the id_order explicitly. In an actual application, this may be done indirectly by matching the id_order to the observation sequence number of the ID variable in a data set.

We now illustrate these concepts with our example. First, we consider the default case in which only the mandatory neighbors dictionary is passed as an argument:

```
>>> w1 = pysal.W(neighbors)

WARNING: there is one disconnected observation (no neighbors)
Island id:  [9]
```

As expected, this generates a warning message that the observation with ID = 9 has no neighbors. While we defer a full discussion of the attributes of the spatial weights object to the next section, we consider three core characteristics at this point: the number of observations, n; a dictionary containing the spatial weights, weights; and a list with the order for the IDs, id_order. In our example, this gives:

```
>>> w1.n
10
>>> w1.weights
{0: [1.0, 1.0],
 1: [1.0, 1.0, 1.0],
 2: [1.0, 1.0],
 3: [1.0, 1.0, 1.0],
 4: [1.0, 1.0, 1.0, 1.0],
 5: [1.0, 1.0, 1.0],
 6: [1.0, 1.0],
 7: [1.0, 1.0, 1.0],
 8: [1.0, 1.0],
 9: [ ]}
>>> w1.id_order
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The default value for the weights is 1.0. They are listed in the lexicographic order for the ID variables, which is also the id_order. In our example, this happens to be the order in which we entered the IDs in the `neighbors` dictionary. However, the two are unconnected. To illustrate this, consider the same neighbors dictionary, but now with entries in column-wise order, as:

```
>>> neighbors1 = { 9 : [ ],
              0: [3, 1], 3: [ 0, 6, 4], 6: [3, 7],
              1: [0, 4, 2], 4: [1, 3, 7, 5], 7: [4, 6, 8],
              2: [1,5], 5: [2,4,8], 8: [5, 7]}
>>> w1a = pysal.W(neighbors1)

WARNING: there is one disconnected observation (no neighbors)
Island id:  [9]
```

The resulting `weights` dictionary does not follow the order given in `neighbors1`, but the same lexicographic order as before.[5]

```
>>> w1a.weights
{0: [1.0, 1.0],
 1: [1.0, 1.0, 1.0],
 2: [1.0, 1.0],
 3: [1.0, 1.0, 1.0],
 4: [1.0, 1.0, 1.0, 1.0],
 5: [1.0, 1.0, 1.0],
 6: [1.0, 1.0],
 7: [1.0, 1.0, 1.0],
 8: [1.0, 1.0],
 9: [ ]}
>>> w1a.id_order
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We now set the weights explicitly, and create a dictionary with values that correspond to row-standardized weights:

```
>>> myweights = {0: [0.5, 0.5],
                1: [0.3333, 0.3333, 0.3333],
                2: [0.5, 0.5],
                3: [0.3333, 0.3333, 0.3333],
                4: [0.25, 0.25, 0.25, 0.25],
                5: [0.3333, 0.3333, 0.3333],
                6: [0.5, 0.5],
                7: [0.3333, 0.3333, 0.3333],
                8: [0.5, 0.5],
                9: [ ]}
```

With the `weights` argument specified and using the original `neighbors` dictionary, the new weights object is then:

```
>>> w2 = pysal.W(neighbors,weights=myweights)
```

[5]Recall that in Python a dictionary does not have an inherent order.

```
WARNING: there is one disconnected observation (no neighbors)
Island id:  [9]
```

which contains:

```
>>> w2.weights
{0: [0.5, 0.5],
 1: [0.3333, 0.3333, 0.3333],
 2: [0.5, 0.5],
 3: [0.3333, 0.3333, 0.3333],
 4: [0.25, 0.25, 0.25, 0.25],
 5: [0.3333, 0.3333, 0.3333],
 6: [0.5, 0.5],
 7: [0.3333, 0.3333, 0.3333],
 8: [0.5, 0.5],
 9: [ ]}
```

Next, we explicitly change the order of `IDs` to reflect a column-wise sequence and pass the associated list as an argument to the weights class constructor:

```
>>> order_id = [ 9, 0, 3, 6, 1, 4, 7, 2, 5, 8 ]
>>> w3 = pysal.W(neighbors,weights=myweights,id_order=order_id)
```

This yields:

```
>>> w3.weights
{0: [0.5, 0.5],
 1: [0.3333, 0.3333, 0.3333],
 2: [0.5, 0.5],
 3: [0.3333, 0.3333, 0.3333],
 4: [0.25, 0.25, 0.25, 0.25],
 5: [0.3333, 0.3333, 0.3333],
 6: [0.5, 0.5],
 7: [0.3333, 0.3333, 0.3333],
 8: [0.5, 0.5],
 9: [ ]}

>>> w3.id_order
 [9, 0, 3, 6, 1, 4, 7, 2, 5, 8]
```

The contents of the `weights` dictionary are still given with the keys in lexicographic order, even though the `id_order` is different. The latter will be used to match the weights to observations in a data set (i.e., a `numpy` array), for example, in the calculation of a spatially lagged variable. However, it is immaterial in the way the contents of the `weights` dictionary are listed.

Finally, with `silent_island_warning` set to `True`, we do not receive a warning after creating a weights object with isolates:

```
>>> w4 = pysal.W(neighbors,weights=myweights,
                 id_order=order_id,
                 silent_island_warning=True)
```

So, no warning, which we confirm by checking the value of the corresponding attribute (`silent_island_warning`):

```
>>> w4.silent_island_warning
True
```

3.4.1.2 Attributes of a Spatial Weights Object

The `PySAL` spatial weights object has a total of 37 attributes, including the four arguments used in its construction (`neighbors`, `weights`, `id_order` and `silent_island_warning`). The attributes can be broadly classified into four groups:

- basic descriptors of the weights object
- statistical characteristics of the spatial weights
- transformation functions
- auxiliary variables used in the computation of test statistics and estimators

We briefly review them in turn, in alphabetical order, by group.

Basic Descriptors of the Weights Object
To fully appreciate the properties of the weights object, it is important to make a distinction between three important aspects: the `ID`, the value of the weight, and the sequence number of the `ID` in the `id_order` list. For example, using the previously created object `w3`, we see that the `ID` for observation `0` is in position `1` in the `id_order` list. Below, we give the main attributes that describe the weights object with a brief explanation and illustration using the `w3` example. For attributes that can also be passed as arguments, we refer to the previous Section.

- `id2i`: a dictionary with as key the `ID` and as value the sequence number (starting with `0`) of that `ID` in the `id_order` list

  ```
  >>> w3.id2i
  {0: 1, 1: 4, 2: 7, 3: 2, 4: 5, 5: 8, 6: 3, 7: 6, 8: 9,
   9: 0}
  ```

- `id_order`: an argument (see previous Section)
- `id_order_set`: a Boolean flag indicating whether or not the `id_order` was set explicitly; the default is `False`, but since we did set this argument for `w3`, the result is

```
>>> w3.id_order_set
True
```

- `neighbor_offsets`: a dictionary with as key the `ID` and as value a list with the sequence numbers of the neighbors from the `id_order` list; this dictionary has the same structure as the `neighbors` dictionary, but the neighbor `ID`s are replaced by their sequence numbers

```
>>> w3.neighbor_offsets
{0: [2, 4],
 1: [1, 5, 7],
 2: [4, 8],
 3: [1, 3, 5],
 4: [4, 2, 6, 8],
 5: [7, 5, 9],
 6: [2, 6],
 7: [5, 3, 9],
 8: [8, 6],
 9: []}
```

- `neighbors`: the mandatory argument to `W` (see previous Section)
- `silent_island_warning`: an argument (see previous Section)
- `transform`: the current transformation, is originally set as `'O'`; the transformations are the same as for `GeoDaSpace` (see Section 3.3.4.1)
- `transformations`: a nested dictionary with as key the transformations that were carried out and as value the corresponding `weights` dictionary; this records all previous transformations so that one can quickly move back without recalculating the transformation

```
>>> w3.transform = 'B'
>>> w3.transformations
{'B': {0: [1.0, 1.0],
  1: [1.0, 1.0, 1.0],
  2: [1.0, 1.0],
  3: [1.0, 1.0, 1.0],
  4: [1.0, 1.0, 1.0, 1.0],
  5: [1.0, 1.0, 1.0],
  6: [1.0, 1.0],
  7: [1.0, 1.0, 1.0],
  8: [1.0, 1.0],
  9: [ ]},
 'O': {0: [0.5, 0.5],
  1: [0.3333, 0.3333, 0.3333],
    2: [0.5, 0.5],
    3: [0.3333, 0.3333, 0.3333],
    4: [0.25, 0.25, 0.25, 0.25],
    5: [0.3333, 0.3333, 0.3333],
    6: [0.5, 0.5],
    7: [0.3333, 0.3333, 0.3333],
    8: [0.5, 0.5],
    9: [ ]}}
```

- `weights`: the weights dictionary (see previous Section)

Statistical Characteristics of the Spatial Weights
The statistical characteristics of the spatial weights are the same characteristics reviewed for `GeoDaSpace` in Section 3.3.4, with a few additions.

- `asymmetries`: a list of `ID` pairs for which the weights are asymmetric, i.e., i is a neighbor of j, but j is not a neighbor of i; for contiguity weights, this list will always be empty, unless there was an error in the `neighbors` dictionary

```
>>> w3.asymmetries
[ ]
```

- `cardinalities`: a dictionary with as key the observation `ID` and as value the number of neighbors for that observation

```
>>> w3.cardinalities
{0: 2, 1: 3, 2: 2, 3: 3, 4: 4, 5: 3, 6: 2, 7: 3, 8: 2,
 9: 0}
```

- `histogram`: a list of tuples with as first element the number of neighbors (cardinality) and as second element the number of observations with that many neighbors (i.e., the same as what is visualized by the connectivity histogram in `GeoDa`)

```
>>> w3.histogram
[(0, 1), (1, 0), (2, 4), (3, 4), (4, 1)]
```

- `islands`: a list with the `ID`s of the unconnected observations

```
>>> w3.islands
[9]
```

- `max_neighbors`: the maximum number of neighbors

```
>>> w3.max_neighbors
4
```

- `mean_neighbors`: the average number of neighbors

```
>>> w3.mean_neighbors
2.3999999999999999
```

- `min_neighbors`: the minimum number of neighbors (will be `0` if there are unconnected observations

```
>>> w3.min_neighbors
0
```

- `n`: the number of observations

```
>>> w3.n
10
```

- `nonzero`: the number of non-zero cells in the $n \times n$ weights matrix

```
>>> w3.nonzero
24
```

- `pct_nonzero`: the percent non-zero cells in the $n \times n$ weights matrix

```
>>> w3.pct_nonzero
0.24
```

- `sd`: the standard deviation of the number of neighbors

```
>>> w3.sd
1.019803902718557
```

Transformation Functions
The transformation methods convert weights objects from one format to another, or handle other utility functions.

- `asymmetry`: method used to compute the asymmetries attribute, returns the list of asymmetries, or an empty list, if none are present (the default for proper contiguity weights)

```
>>> w3.asymmetry( )
[ ]
```

- `full`: creates a full `numpy` array from the weights information; returns a tuple with the full matrix as the first element and the `id_order` as the second element

```
>>> w3.full( )
(array([[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 1., 0., 1., 0., 0., 0., 0., 0.],
       [ 0., 1., 0., 1., 0., 1., 0., 0., 0., 0.],
       [ 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.],
       [ 0., 1., 0., 0., 0., 1., 0., 1., 0., 0.],
       [ 0., 0., 1., 0., 1., 0., 1., 0., 1., 0.],
       [ 0., 0., 0., 1., 0., 1., 0., 0., 0., 1.],
       [ 0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
       [ 0., 0., 0., 0., 0., 1., 0., 1., 0., 1.],
       [ 0., 0., 0., 0., 0., 0., 1., 0., 1., 0.]]),
 [9, 0, 3, 6, 1, 4, 7, 2, 5, 8])
```

If one just wants the full array, the function needs to be subscripted by (`[0]`) as in

```
>>> w3.full( )[0]
array([[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
      [ 0., 0., 1., 0., 1., 0., 0., 0., 0., 0.],
      [ 0., 1., 0., 1., 0., 1., 0., 0., 0., 0.],
      [ 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.],
      [ 0., 1., 0., 0., 0., 1., 0., 1., 0., 0.],
      [ 0., 0., 1., 0., 1., 0., 1., 0., 1., 0.],
      [ 0., 0., 0., 1., 0., 1., 0., 0., 0., 1.],
      [ 0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
      [ 0., 0., 0., 0., 0., 1., 0., 1., 0., 1.],
      [ 0., 0., 0., 0., 0., 0., 1., 0., 1., 0.]])
```

- `get_transform`: method to return the current weights transformation, same effect as the `transform` attribute

```
>>> w3.get_transform( )
'B'
```

- `set_shapefile`: method to record the items for a `GAL` header file, for internal use only

- `set_transform`: method to set weights transformation to one of '`O`', '`B`', '`R`', or '`V`'; same effect as setting the `transform` attribute directly

```
>>> w3.transform = 'B'
>>> w3.set_transform('R')
>>> w3.transform
'R'
```

- `sparse`: an attribute of the spatial weights object that contains a sparse `numpy` array representation of the weights, intended for manipulation as an array (not as a sparse spatial weights object, see `towsp`)

```
>>> w3.sparse
<10x10 sparse matrix of type '<type 'numpy.float64'>'
with 24 stored elements in Compressed Sparse Row format>
```

- `towsp`: method to convert a regular spatial weights object to a sparse spatial weights object (class `WSP`); the result is not just a sparse `numpy` array (as for `sparse`), but an actual weights object with attributes

```
>>> ws = w3.towsp( )
>>> ws
<pysal.weights.weights.WSP at 0xb266930>
```

Auxiliarly Variables
A series of auxiliary variables are stored as attributes of a spatial weights object to facilitate the calculation of tests and diagnostics for spatial autocorrelation, such as Moran's I and the Lagrange Multiplier tests (see Section 5.1.6 of Chapter 5). Typically, these are not needed in and of themselves, but they can be investigated for pedagogical purposes, e.g., to illustrate the different components of the statistical inference, such as expression $tr(\mathbf{WW} + \mathbf{W'W})$ in Equation 5.15.

- `diagW2`: the elements of the diagonal of the matrix $\mathbf{WW}$

```
>>> w3.diagW2
array([ 0.        ,  0.33333333,  0.41666667,  0.33333333,
        0.41666667, 0.33333333,  0.41666667,  0.33333333,
        0.41666667,  0.33333333])
```

- `diagWtW`: the elements of the diagonal of the matrix $\mathbf{W'W}$

```
>>> w3.diagWtW
array([ 0.        ,  0.22222222,  0.5625    ,  0.22222222,
        0.5625    ,  0.44444444,  0.5625    ,  0.22222222,
        0.5625    ,  0.22222222])
```

- `diagWtW_WW`: the elements of the diagonal of the sum of matrices $\mathbf{WW}$ and $\mathbf{W'W}$ (i.e., the sum of the matching elements of `diagW2` and `diagWtW`)

```
>>> w3.diagWtW_WW
array([ 0.        ,  0.55555556,  0.97916667,  0.55555556,
        0.97916667,  0.77777778,  0.97916667,  0.55555556,
        0.97916667,  0.55555556])
```

- `s0`: the sum of all weights $S_0 = \sum_i \sum_j w_{ij}$; note that in our example this is one less than the number of observations (9 = 10 - 1) because of the unconnected unit, in normal circumstances $S_0 = n$ for row-standardized weights

```
>>> w3.s0
8.9999999999999982
```

- s1: $S_1 = \sum_i \sum_j (w_{ij} + w_{ji})^2$; not used in spatial econometrics

```
>>> w3.s1
6.9166666666666661
```

- s2: $S_2 = \sum_j (\sum_i w_{ij} + \sum_i w_{ji})^2$; not used in spatial econometrics

```
>>> w3.s2
36.80555555555555
```

- s2array: an array that constitutes an intermediate step in the computation of s2, consisting of the elements for each j (s2 is the sum of the elements in this array); not used in spatial econometrics

```
>>> w3.s2array
array([[ 0.        ],
       [ 2.77777778],
       [ 5.0625    ],
       [ 2.77777778],
       [ 5.0625    ],
       [ 5.44444444],
       [ 5.0625    ],
       [ 2.77777778],
       [ 5.0625    ],
       [ 2.77777778]])
```

- trcW2: the trace of the matrix $\mathbf{WW}$, i.e., the sum of the elements of diagW2

```
>>> w3.trcW2
3.3333333333333335
```

- trcWtW: the trace of the matrix $\mathbf{W'W}$, i.e., the sum of the elements of diagWtW

```
>>> w3.trcWtW
3.5833333333333335
```

- trcWtW_WW: the trace of the matrix $\mathbf{WW} + \mathbf{W'W}$, i.e., the sum of trcW2 and trcWtW

```
>>> w3.trcWtW_WW
6.9166666666666661
```

3.4.1.3 The Sparse Spatial Weights Class `WSP`

In `spreg`, all the weights manipulations are carried out using sparse spatial weights objects. These are constructed under the hood and seldom need to be created explicitly. For completeness sake, we briefly describe the sparse spatial weights class `WSP` and its attributes.

We already saw one way to create a `WSP` object from a regular spatial weights object by means of the `towsp` method (see previous Section). The way to create such an object directly is:

```
>>> ws = pysal.WSP(sparse,id_order=None)
```

with as arguments:

- `sparse`: a matrix in a `scipy` sparse array format; required argument
- `id_order`: optional, the `id_order` list, same as for a regular spatial weights object

The sparse array passed to `WSP` can be any sparse format supported by `scipy`. Internally, it is converted to a CSR (compressed sparse row) format, so this extra step can be avoided by passing a CSR array to begin with.

We illustrate this by continuing to use our previous example, where we created the weights object `w3`. We saw in the previous Section how we can extract a sparse array from this object using the `sparse` attribute. This array is in the CSR format, so we can use it as an input to `WSP`.

```
>>> sparsemat = w3.sparse
>>> sparsemat
<10x10 sparse matrix of type '<type 'numpy.float64'>'
with 24 stored elements in Compressed Sparse Row format>
>>> sparseW = pysal.weights.WSP(sparsemat,id_order=order_id)
>>> sparseW
<pysal.weights.weights.WSP at 0xb2f9490>
```

Of course, in practice, one will typically first create a regular spatial weights object and then convert it to a sparse object using the `towsp` method. Finally, a sparse spatial weights object can be converted to a regular spatial weights object by means of the `pysal.weights.WSP2W` function (see the online API reference for technical details).

3.4.1.4 Attributes of a Sparse Spatial Weights Object

Sparse spatial weights objects have only a small subset of the attributes of a regular weights object. They are intended to be manipulated directly as sparse matrices, so the attributes are limited to the arguments passed, `id_order`, the number of observations, `n`, and two auxiliary variables, `s0` and `trcWtW_WW`. Their meaning is the same as for a regular spatial weights object.

In our example:

```
>>> sparseW.id_order
[9, 0, 3, 6, 1, 4, 7, 2, 5, 8]
>>> sparseW.n
10
>>> sparseW.s0
8.9999999999999982
>>> sparseW.trcWtW_WW
6.9166666666666661
```

The other attributes of a weights object are not available and will generate an error message when requested. For example:

```
>>> sparseW.transform
AttributeError   Traceback (most recent call last)
...
----> 1 sparseW.transform
AttributeError: 'WSP' object has no attribute 'transform'
```

3.4.2 Creating Contiguity Weights from a Shapefile

Spatial weights objects based on the contiguity between polygons can be created directly from ESRI shape files. `PySAL` currently supports both `queen` and `rook`, as well as higher order contiguity.[6]

3.4.2.1 Queen Contiguity Weights

In `PySAL`, queen contiguity weights are constructed by means of the special user function `queen_from_shapefile` . The only mandatory argument for this function is the filename for the shape file, including the `shp` file extension. When the shape file is not present in the current working directory, the full pathname needs to be specified. The complete call is:

```
>>> w = pysal.queen_from_shapefile(shapefile, idVariable=None,
        sparse=False)
```

The optional arguments are:

- `idVariable`: a variable from the data base associated with the shape file (the `dbf` file) to be used as an `ID` variable for the weights

[6]The contiguity extraction algorithms in `PySAL` assume the shapefile respects planar enforcement, requiring that no polygons overlap and that any lines must be split at points of intersection.

- `sparse`: a boolean flag to indicate whether a regular spatial weights object is created (`False`, the default) or a sparse spatial weights object (`True`)

The function returns a spatial weights object, either a regular one (the default), or a sparse spatial weights object for `sparse = True`. These objects have all the attributes discussed previously, respectively in Section 3.4.1.2 for regular weights and in Section 3.4.1.4 for sparse weights.

To illustrate this, we again take the U.S. county `NAT.shp` data from the `PySAL` examples set. We use the `pysal.examples.get_path` function to make sure the proper path name is prefixed to the name of the shape file. We set the `idVariable` to `FIPSNO` (this is the same approach as taken for `GeoDa` and `GeoDaSpace`). The full call is then:

```
>>> wq = pysal.queen_from_shapefile(
          pysal.examples.get_path('NAT.shp'),
          idVariable='FIPSNO')
>>> wq
<pysal.weights.weights.W at 0xb3c2510>
```

It is clear that `wq` is a spatial weights object.

It is important to keep in mind that the spatial object created from the shape file has *binary* weights. This is not directly obvious, since the `transform` attribute is initially set to `'O'`:

```
>>> wq.transform
'O'
```

but the weights are binary, for, example, for observation with `FIPSNO = 27077`:

```
>>> wq.weights[27077]
[1.0, 1.0, 1.0]
```

It is therefore critical to *always* follow the creation of the weights object by an explicit row-standardization:

```
>>> wq.transform = 'R'
>>> wq.weights[27077]
[0.3333333333333333, 0.3333333333333333, 0.3333333333333333]
```

With the `sparse` flag set to `True`, the resulting object is a sparse spatial weights object. For example:

```
>>> wqsparse = pysal.queen_from_shapefile(
                 pysal.examples.get_path('NAT.shp'),
                 idVariable='FIPSNO',sparse=True)
>>> wqsparse
<pysal.weights.weights.WSP at 0x9a67830>
```

Note that these weights are binary:

```
>>> wqsparse.s0
18168.0
```

If the weights were row-standardized, `s0` should equal the number of observations, `n = 3085`, so clearly in this case, they are not. In order to get sparse spatial weights objects into row-standardize form, a better approach is to create a regular spatial weights object, say `w`, convert it to row-standardized form using `w.transform = 'R'` and then obtain a sparse spatial weights object from `w.towsp( )`.

3.4.2.2 Rook Contiguity Weights

Rook contiguity weights are constructed from a shape file by means of the user function `rook_from_shapefile`. The complete call is identical to that for queen weights, except that a different function name is used:

```
>>> wr = pysal.rook_from_shapefile(shapefile, idVariable=None,
         sparse=False)
```

The arguments and options are the same as those covered in Section 3.4.2.1, and they will not be repeated here.

3.4.3 Creating Weights for a Regular Lattice Structure

`PySAL` includes functionality to generate a spatial weights object directly for any rectangular or hexagonal lattice structure.[7] This is often very useful in the context of simulation experiments. Since the neighbor structure of the lattice is known, there is no need to construct it by reading the contents of a shape file.

For a rectangular lattice, the function is `lat2W`, invoked as:

```
>>> wgrid = pysal.lat2W(nrows=5,ncols=5,rook=True,id_type='int')
```

This function returns a standard spatial weights object.[8] Without any arguments, i.e., `pysal.lat2W( )`, this results in the weights for a 5×5 regular grid, i.e., a 25×25 weights matrix. The four arguments are:

- `nrows`: the number of rows in the grid, default is `5`
- `ncols`: the number of columns in the grid, default is `5`
- `rook`: the type of contiguity, default is `True` for rook contiguity

[7]Note that this functionality is not available through the GUI of `GeoDa` or `GeoDaSpace`.

[8]`PySAL` also contains a function that creates a sparse spatial weights object for a regular lattice: `pysal.weights.lat2SW`. See the online API reference for details.

- `id_type`: the representation of the `ID` variable, either as an integer, starting with `0` (`'int'`), as a real variable, starting with `0.0` (`'float'`), or as a simple string, starting with `'id0'` (`'string'`)

We return to the 3×3 lattice we used as an example before, and explore the three options for the `id_type`. First, the default case, using integer values:

```
>>> w3x3 = pysal.lat2W(3,3)
>>> w3x3.weights
{0: [1.0, 1.0],
 1: [1.0, 1.0, 1.0],
 2: [1.0, 1.0],
 3: [1.0, 1.0, 1.0],
 4: [1.0, 1.0, 1.0, 1.0],
 5: [1.0, 1.0, 1.0],
 6: [1.0, 1.0],
 7: [1.0, 1.0, 1.0],
 8: [1.0, 1.0]}
```

next, using real valued `ID`s (the key in the weights dictionary):

```
>>> w3x3f = pysal.lat2W(3,3,id_type='float')
>>> w3x3f.weights
{0.0: [1.0, 1.0],
 1.0: [1.0, 1.0, 1.0],
 2.0: [1.0, 1.0],
 3.0: [1.0, 1.0, 1.0],
 4.0: [1.0, 1.0, 1.0, 1.0],
 5.0: [1.0, 1.0, 1.0],
 6.0: [1.0, 1.0],
 7.0: [1.0, 1.0, 1.0],
 8.0: [1.0, 1.0]}
```

and finally, for strings:

```
>>> w3x3s = pysal.lat2W(3,3,id_type='string')
>>> w3x3s.weights
{'id0': [1.0, 1.0],
 'id1': [1.0, 1.0, 1.0],
 'id2': [1.0, 1.0],
 'id3': [1.0, 1.0, 1.0],
 'id4': [1.0, 1.0, 1.0, 1.0],
 'id5': [1.0, 1.0, 1.0],
 'id6': [1.0, 1.0],
 'id7': [1.0, 1.0, 1.0],
 'id8': [1.0, 1.0]}
```

A similar function for hexagonal grids is `pysal.hexLat2W`. For further technical details, we refer to the online API reference.

3.4.4 Block Weights

A particular form of contiguity weights are the block weights introduced in Section 3.1.2. `PySAL` supports the creation of these weights, but neither `GeoDa` or `GeoDaSpace` currently do. The function that accomplishes this is `regime_weights`. The full call is:

```
>>> wreg = pysal.regime_weights(regimes)
```

with one required argument:

- `regimes`: a list or `numpy` array matching the observations, with for each observation the regime it belongs to.

The identifiers for the regimes can be numeric, unique integers or unique floats, or strings (unique for each regime). The result is a spatial weights object in binary form (i.e., *not* with row-standardized weights).

To illustrate this, consider a list for a data set with 15 observations and three regimes, indicated by `'s'`, `'e'` and `'w'`. The list is:

```
>>> regimes = ['s', 's', 's', 's', 's',
               'e', 'e', 'e', 'e', 'e',
               'w', 'w', 'w', 'w', 'w']
```

The block weights are created as:

```
>>> wreg = pysal.regime_weights(regimes)
```

with the following neighbor structure:

```
>>> wreg.neighbors
{0: [1, 2, 3, 4],
 1: [0, 2, 3, 4],
 2: [0, 1, 3, 4],
 3: [0, 1, 2, 4],
 4: [0, 1, 2, 3],
 5: [6, 7, 8, 9],
 6: [5, 7, 8, 9],
 7: [5, 6, 8, 9],
 8: [5, 6, 7, 9],
 9: [5, 6, 7, 8],
 10: [11, 12, 13, 14],
 11: [10, 12, 13, 14],
 12: [10, 11, 13, 14],
 13: [10, 11, 12, 14],
 14: [10, 11, 12, 13]}
```

Each of the observations is a neighbor to the other observations in the same block, but not to any outside the block.

3.4.5 Higher Order Contiguity

Once a first order contiguity spatial weights object is available, higher order contiguity can be obtained in a straightforward manner. The PySAL higher_order function takes as arguments a weights object and the order of contiguity (as an integer).[9] The complete call is:

```
w_high = pysal.higher_order(w, k = 2)
```

where

- w: a regular (i.e., not sparse) spatial weights object, required
- k: the order of contiguity, default k = 2

This returns a spatial weights object with the higher order contiguity, using the same Anselin and Smirnov (1996) algorithm as GeoDa. However, in contrast to the implementation in GeoDa and GeoDaSpace, there is no option to include lower order neighbors. This must be carried out explicitly by means of the PySAL w_union function:

```
w_combined = pysal.w_union(w1, w2,
             silent_island_warning = False)
```

The required arguments to this function are two (non-sparse) spatial weights objects. Optional is a flag for the silent_island_warning which has the same meaning as in the construction of a spatial weights object (see Section 3.4.1.1). It returns a new spatial weights object (with binary weights) in which the neighbor structures of the two arguments are combined (union). In the special case of higher order contiguity weights, this must be implemented in multiple steps, gradually building up the weights structure until the desired order is achieved. For example, if second order contiguity with lower order neighbors is desired, this is accomplished by applying w_union to the first and second order weights. However, for higher orders, such as third, the first and second need to be combined first, then the combined weights need to be merged with the third order contiguity, in a step-wise fashion.

We illustrate this for the queen contiguity weights we created in Section 3.4.2.1. Second order contiguity follows as:

```
>>> wq2 = pysal.higher_order(wq,k=2)
```

and including the first order neighbors from:

```
>>> wq2include = pysal.w_union(wq,wq2)
```

[9]The PySAL function pysal.weights.higher_order_sp implements higher order contiguity for sparse weights objects. See the online API reference for technical details.

We now compare the neighbors for observation with `FIPSNO = 53019` in the first order contiguity weights and the two higher order weights (see also lines `3-4` in the `GAL` file of Figure 3.5 for the first order neighbors of `FIPSNO = 53019`):

```
>>> wq.neighbors[53019]
[53065, 53043, 53047]
>>> wq2.neighbors[53019]
[53071, 53077, 12131, 4001, 4025, 45003, 45009, 45021, 45025]
>>> wq2include.neighbors[53019]
[4001, 12131, 45025, 53065, 45003, 53071, 45009,
 53043, 53077, 53047, 4025, 45021]
```

As expected, the neighbors for `wq2include` combine the lists for the first and second order contiguity weights.

3.4.6 Reading Weights Files

Arguably, the most straightforward way to create a spatial weights object in `PySAL` is to read its content from a pre-existing file. As previously shown in Figure 3.14, `PySAL` currently supports 11 weights file formats. Reading a weights file operates in the same manner as reading any other file in `PySAL` and is based on the `FILEIO` module. It operates in three basic steps: (i) create a file handle to open the file for reading; (ii) read the contents and convert into a spatial weights object; and (iii) close the file.

We illustrate this using the `nat_queen.gal` queen contiguity file for the U.S. counties that we created with `GeoDa` in Section 3.2.2.1. A partial glance at its contents is given in Figure 3.5.

The three steps to read this `GAL` file into a spatial weights object are:

```
>>> galw = pysal.open(pysal.examples.get_path('nat_queen.gal'),
                      'r')
>>> w = galw.read()
>>> galw.close()
```

We now have the spatial weights object `w` and we can check its attributes in the usual fashion. For example, the dimension is found as:

```
>>> w.n
3085
```

The neighbors and weights for `ID = '27077'` are:

```
>>> w.neighbors['27077']
['27007', '27135', '27071']
>>> w.weights['27077']
[1.0, 1.0, 1.0]
```

Note that the weights are binary, so that an explicit row-standardization always needs to be carried out before using the weights in spatial regression operations. Also, the `ID` variable is interpreted as a string, rather than an integer, hence the need to surround the value by quotes. It is good practice to check the nature of the `ID` variable by listing part of the `id_order` attribute. For example:

```
>>> w.id_order[:3]
['27077', '53019', '53065']
```

Weights files created in other formats are read in the same fashion, by using the `open` and `read` commands on the file (with full path name specified if needed) with the `'r'` option. `PySAL` internally recognizes the file extension and uses the appropriate file reader for that format.

3.4.7 Writing and Converting Weights Files

Weights objects created in `PySAL` can be written out to files in a range of supported formats (see Figure 3.14). The file extension determines the format that will be used. For example, assuming we created the file object `w` by reading from the file `nat_queen.gal` (as in Section 3.4.6), its contents can be written to a `GAL` format file by means of the `open` and `write` commands on the file (with full path name specified if needed), with the `'w'` option and the weights object (`w`) specified as the argument of the `write` function. In our example, using `natqueen1.gal` as the file name for the output file, this is accomplished with:

```
>>> galwout = pysal.open("natqueen1.gal",'w')
>>> galwout.write(w)
>>> galwout.close()
```

The first few lines of the `natqueen1.gal` file are as follows:

```
3085
27077 3
27007 27135 27071
53019 3
53047 53065 53043
53065 4
53043 53051 53063 53019
```

Note how the original header line is replaced by a single item, the number of observations (`3085`). In all other respects, the file is identical to the input file.

`PySAL` also contains functionality to convert between weights formats directly, using the `pysal.weight_convert` command. However, this will typically not be necessary, since a weights object can be read from any supported format and subsequently written to a file in any