

Data and Spatial Weights in spdep Notes and Illustrations

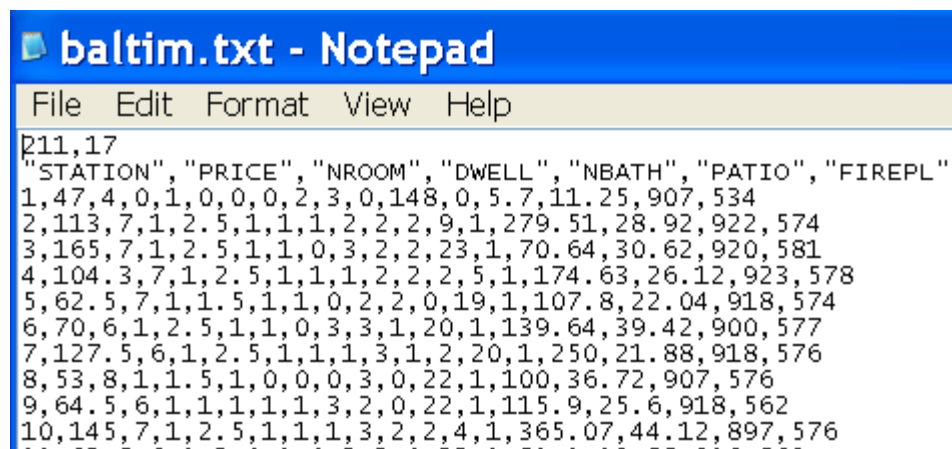
Luc Anselin
University of Illinois, Urbana-Champaign
<http://sal.agecon.uiuc.edu>
July 11, 2003

Purpose

The purpose of this set of notes is to illustrate how to bring data and spatial weights information from GeoDa into R, for use by the spdep package. In addition, several of the weights construction and manipulation functions in spdep are reviewed. Both built-in spdep functions as well as a smaller number of customized R functions will be used and illustrated. For details on GeoDa functionality, see the GeoDa User's Guide and Tutorials at <http://sal.agecon.uiuc.edu/stuff>. Further details on the operations of spdep can be found in the original spdep documentation and in the "Introduction to Spatial Regression Analysis in R" tutorial on the SAL web site. The various tasks will be illustrated with a sample data set. It is assumed that you will try to replicate this with your own data and/or with another one of the sample data sets. To get started, make sure the spdep package is loaded, with `library(spdep)`

Converting GeoDa data to a data frame

Data in R are most usefully stored in a data frame. The helper function `read.geoda` uses the built-in `read.csv` function with some preset options to make sure it matches the format of the data exported by GeoDa. For example, load the `BALTIM.SHP` point data set into GeoDa and use `Tools > Data Export > ASCII`. Specify the input shape file and give a name for the text file that will contain the exported data (such as `baltim.txt`). You will have exported a file that looks like Figure 1.



```
baltim.txt - Notepad
File Edit Format View Help
11,17
"STATION", "PRICE", "NROOM", "DWELL", "NBATH", "PATIO", "FIREPL",
1,47,4,0,1,0,0,0,2,3,0,148,0,5.7,11.25,907,534
2,113,7,1,2.5,1,1,1,2,2,2,9,1,279.51,28.92,922,574
3,165,7,1,2.5,1,1,0,3,2,2,23,1,70.64,30.62,920,581
4,104.3,7,1,2.5,1,1,1,2,2,2,5,1,174.63,26.12,923,578
5,62.5,7,1,1.5,1,1,0,2,2,0,19,1,107.8,22.04,918,574
6,70,6,1,2.5,1,1,0,3,3,1,20,1,139.64,39.42,900,577
7,127.5,6,1,2.5,1,1,1,3,1,2,20,1,250,21.88,918,576
8,53,8,1,1.5,1,0,0,0,3,0,22,1,100,36.72,907,576
9,64.5,6,1,1,1,1,1,3,2,0,22,1,115.9,25.6,918,562
10,145,7,1,2.5,1,1,1,3,2,2,4,1,365.07,44.12,897,576
```

Figure 1. Contents of baltim.txt, exported by GeoDa

Two things are non-standard as far as the usual R function `read.table` goes. One is the extra line in the front (with number of observations and number of variables), the other is the use of a comma as data separator. You can deal with these by explicitly setting the “skip” option to 1 and using `read.csv` instead of the generic `read.table`. For example:

```
> balt <- read.csv("baltim.txt",header=TRUE,skip=1)
> summary(balt)
```

STATION		PRICE		NROOM		DWELL	
Min.	: 1.0	Min.	: 3.50	Min.	: 3.000	Min.	:0.0000
1st Qu.:	53.5	1st Qu.:	30.95	1st Qu.:	5.000	1st Qu.:	0.0000
Median :	106.0	Median :	40.00	Median :	5.000	Median :	1.0000
Mean :	106.0	Mean :	44.31	Mean :	5.199	Mean :	0.5355
3rd Qu.:	158.5	3rd Qu.:	53.75	3rd Qu.:	6.000	3rd Qu.:	1.0000
Max.	:211.0	Max.	:165.00	Max.	:10.000	Max.	:1.0000

etc...

You can also explicitly specify a record identifier by means of the `row.names` option. For the `BALTIM.TXT` data, this would be `STATION`:

```
>balt3 <- read.csv("baltim.txt",header=TRUE,skip=1,row.names="STATION")
```

As a result of this specification, the `STATION` column is no longer part of the data frame as a separate variable, it is now used explicitly as the row indicator.

```
> summary(balt3)
```

PRICE		NROOM		DWELL		NBATH	
Min.	: 3.50	Min.	: 3.000	Min.	:0.0000	Min.	:1.000
1st Qu.:	30.95	1st Qu.:	5.000	1st Qu.:	0.0000	1st Qu.:	1.000
Median :	40.00	Median :	5.000	Median :	1.0000	Median :	1.500
Mean :	44.31	Mean :	5.199	Mean :	0.5355	Mean :	1.573
3rd Qu.:	53.75	3rd Qu.:	6.000	3rd Qu.:	1.0000	3rd Qu.:	2.000
Max.	:165.00	Max.	:10.000	Max.	:1.0000	Max.	:5.000

etc...

These options are incorporated in the `read.geoda` helper function. You “compile” this function with the R `source` command. Make sure you copy the file `read.geoda.R` to your working directory. Then “source” it as

```
> source("read.geoda.R")
```

The `read.geoda` function (drop the R file extension) is now added to your workspace and can be invoked directly. For example, to create a data frame without a row identifier:

```
> balt00 <- read.geoda("baltim.txt")
> summary(balt00)
```

STATION		PRICE		NROOM		DWELL	
Min.	: 1.0	Min.	: 3.50	Min.	: 3.000	Min.	:0.0000
1st Qu.:	53.5	1st Qu.:	30.95	1st Qu.:	5.000	1st Qu.:	0.0000
Median :	106.0	Median :	40.00	Median :	5.000	Median :	1.0000
Mean :	106.0	Mean :	44.31	Mean :	5.199	Mean :	0.5355
3rd Qu.:	158.5	3rd Qu.:	53.75	3rd Qu.:	6.000	3rd Qu.:	1.0000

```
Max. :211.0    Max. :165.00    Max. :10.000    Max. :1.0000
etc...
```

Specifying `STATION` as the row identifier:

```
> balt01 <- read.geoda("baltim.txt", "STATION")
> summary(balt01)
```

PRICE		NROOM		DWELL		NBATH	
Min. :	3.50	Min. :	3.000	Min. :	0.0000	Min. :	1.000
1st Qu.:	30.95	1st Qu.:	5.000	1st Qu.:	0.0000	1st Qu.:	1.000
Median :	40.00	Median :	5.000	Median :	1.0000	Median :	1.500
Mean :	44.31	Mean :	5.199	Mean :	0.5355	Mean :	1.573
3rd Qu.:	53.75	3rd Qu.:	6.000	3rd Qu.:	1.0000	3rd Qu.:	2.000
Max. :	165.00	Max. :	10.000	Max. :	1.0000	Max. :	5.000

```
etc...
```

In order to be able to refer to the variables in a data frame directly by their name, instead of using `framename$variablename`, you can attach the data frame. For example, if you use the `mean` command with `x`, you get an error message:

```
> mean(X)
Error in mean(X) : Object "X" not found
```

After attaching the data frame, this command works fine:

```
> attach(balt01)
> mean(X)
[1] 911.646
```

After you are done using a given data frame, it is good practice to detach it:

```
> detach(balt01)
```

This frees up work space and avoids possible naming conflicts.

Converting GAL weights to neighbor objects – Old Format, Sequence Numbers

Contiguity information on spatial objects is contained in a neighbor object (`nb`) in `spdep`. This contiguity information can be constructed for polygon shape files using the `CreateWeights` function in `GeoDa`. The resulting GAL file (text file with file extension `.gal`) can be read into `spdep` by means of the `read.gal` function. However, care must be taken with the newer format used by `GeoDa`. Before delving into that, consider the “old” format case, where the first line in the file gives the number of observations. The remainder of the file gives, for each observation, the observation ID, the number of neighbors and then on a second line the IDs for the neighbors. For now, we will use a simple sequence number as an ID. For example, using `GeoDa` to create a set of Thiessen polygons for the `BALTIM` point data set (see the `GeoDa` User’s Guide, pp. 26-28), you can create a “rook” type contiguity weights file. Leave the entry for the `ID` variable untouched, as in Figure 2. This will use the sequence numbers as identifiers.

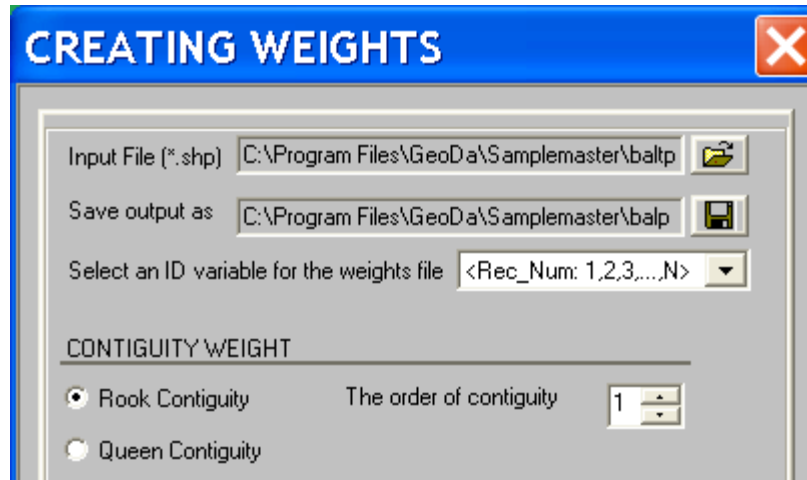


Figure 2. Creating a weights file without explicit ID variable.

This is usually dangerous, unless you can guarantee that the record sequence in the internal storage in GeoDa will be the same as in the external data set. As long as you create the data frame using the exported file from GeoDa, based on the same shape file, things should be OK, but it is “living dangerously.” The explicit designation of an `ID` variable is much preferred.

After creating the GAL file using this procedure, the format of the file is as in Figure 3. The text file has a header line with the number of observations, followed by the contiguity information, going from observation 1 to 211, in sequence.

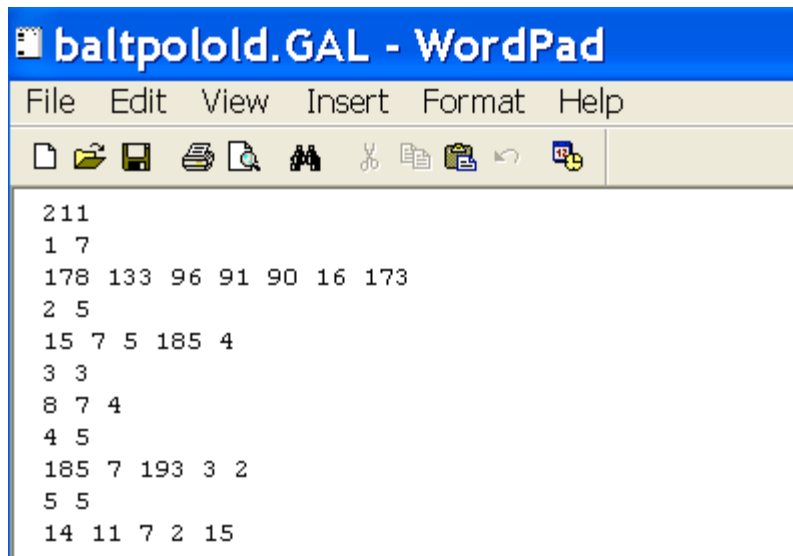


Figure 3. GAL weights file without ID variable.

The standard `read.gal` function in `spdep` assumes that the contiguity information is given in sequence. This has been generalized in the most recent version of `read.gal`

included in the helper files, which is considered in the next section. To have access to the full range of options, it is best to copy the file `read.gal3.R` to your working directory and source it into R:

```
> source("read.gal3.R")
```

Note that this will replace the original `read.gal` function included in the `spdep` package with a more recent (and slightly more flexible) version. It will also add some new functions, such as `read.gal2` and `read.gal3`.

The `read.gal` function has two parameters: the file name (include the GAL file extension), and an option to specify observation ids. The `spdep` manual refers to these as region ids, which is the matching attribute of the `nb` object (`"region.id"`). When no region ID is specified, sequence numbers are assigned. Note that the region ID must be a character vector that is already in the work space (through attaching a data frame, for example). Also, this is not the same as a general identifier for the observations and their neighbors, since its value is not extracted from the GAL file, but taken from the data frame. In the neighbor object, it is added as an additional attribute, in addition to the contiguity information. It is used in the `summary` function, but is not part of the contiguity information. In contrast to the original `read.gal` function in `spdep` version 0.1-10, the newer function can read GAL files with both the old header (only the number of observations) and the new header (four entries). However, it does require that the contiguity entries are in the same sequence as the observations in the data frame.

Using the `baltpolold.GAL` file as the input, a neighbor object is constructed as

```
> baltnb1 <- read.gal("baltpolold.GAL")
```

and its characteristics can be obtained with the `summary` function. Note how the `$region.id` attribute is `chr[1:221]`, a list of sequential integer characters. The summary characteristics describe the connectedness structure of the spatial weights. The “link number distribution” contains the same information as the histogram of weights characteristics in GeoDa (pp. 86-87 in the GeoDa User’s Guide).

```
> summary(baltnb1)
Connectivity of baltnb1 with the following attributes:
List of 5
 $ class      : chr "nb"
 $ region.id  : chr [1:211] "1" "2" "3" "4" ...
 $ gal        : logi TRUE
 $ call       : logi TRUE
 $ sym        : logi TRUE
NULL
Number of regions: 211
Number of nonzero links: 1198
Percentage nonzero weights: 2.690865
Average number of links: 5.677725
Link number distribution:
```

```

3  4  5  6  7  8  9 11
7 21 66 75 29  9  3  1
7 least connected regions:
3 6 48 97 144 157 199 with 3 links
1 most connected region:
208 with 11 links

```

In order to explicitly specify a `region.id` variable, you need to make things a bit more interesting, since `STATION` is a simple sequential numbering. In GeoDa, you can use the table calculation functions to create a new variable, `NEWID`, for example as `1000 + STATION`. Save this as a new shape file, say `baltim2.shp`. Now turn `baltim2.shp` into a new data frame, first exporting the data to `baltim2.txt`, then reading it into a table:

```

> newbalt <- read.geoda("baltim2.txt")
> summary(newbalt)

```

STATION		PRICE		NROOM		DWELL	
Min.	: 1.0	Min.	: 3.50	Min.	: 3.000	Min.	:0.0000
1st Qu.:	53.5	1st Qu.:	30.95	1st Qu.:	5.000	1st Qu.:	0.0000
Median	:106.0	Median	: 40.00	Median	: 5.000	Median	:1.0000
Mean	:106.0	Mean	: 44.31	Mean	: 5.199	Mean	:0.5355
3rd Qu.:	158.5	3rd Qu.:	53.75	3rd Qu.:	6.000	3rd Qu.:	1.0000
Max.	:211.0	Max.	:165.00	Max.	:10.000	Max.	:1.0000

```

...

```

Y		NEWID	
Min.	:505.5	Min.	:1001
1st Qu.:	528.8	1st Qu.:	1054
Median	:544.5	Median	:1106
Mean	:544.2	Mean	:1106
3rd Qu.:	559.0	3rd Qu.:	1159
Max.	:581.0	Max.	:1211

Note the `NEWID` variable. Make sure to detach the old data set and attach the `newbalt` data frame. Now, specify `NEWID` as the id variable (second parameter) in the `read.gal` command and follow this by the `summary` command to obtain the weights characteristics.

```

> attach(newbalt)
> baltnb2 <- read.gal("baltpolold.GAL",NEWID)
> summary(baltnb2)
Connectivity of baltnb2 with the following attributes:
List of 5
 $ class      : chr "nb"
 $ region.id: int [1:211] 1001 1002 1003 1004 1005 1006 1007 1008 1009
1010 ...
 $ gal        : logi TRUE
 $ call       : logi TRUE
 $ sym        : logi TRUE
NULL
Number of regions: 211
Number of nonzero links: 1198
Percentage nonzero weights: 2.690865
Average number of links: 5.677725
Link number distribution:

```

```

3  4  5  6  7  8  9 11
7 21 66 75 29  9  3  1
7 least connected regions:
1003 1006 1048 1097 1144 1157 1199 with 3 links
1 most connected region:
1208 with 11 links

```

Note how the `region.id` is now a list of integer values (not characters) and how the least and most connected regions are identified by their ID value (not by the sequence number). In real applications, these are often more meaningful than simple sequence numbers.

Converting GAL weights to neighbor objects – ID Variable

The new function `read.gal2` is designed to handle GAL format files with contiguity information expressed using a general ID variable. For example, the rook and queen weights for US counties available from the CSISS spatial weights repository use the old header format with ID variables (see <http://sal.agecon.uiuc.edu/weights/index.html>). Note that GeoDa has implemented a slightly more complex header line which can also be parsed correctly by `read.gal2`. To illustrate this function, consider the GAL file shown in Figure 4. This was obtained using the newly created point shape file `baltim2.shp`.

First, a Thiessen polygon file was created that contains the `NEWID` variable, say `baltpoly2`. Then `baltpoly2` was used to create a rook contiguity weights file, specifying `NEWID` as the `ID` variable, as shown in Figure 5. The file in Figure 4 resulted after the original GAL file was edited by hand to reflect the old header line format. The new header line contained the values `0 211 baltpoly0 NEWID`, as shown in Figure 6. All this editing can be carried out in a standard text editor, such as Notepad or Wordpad.

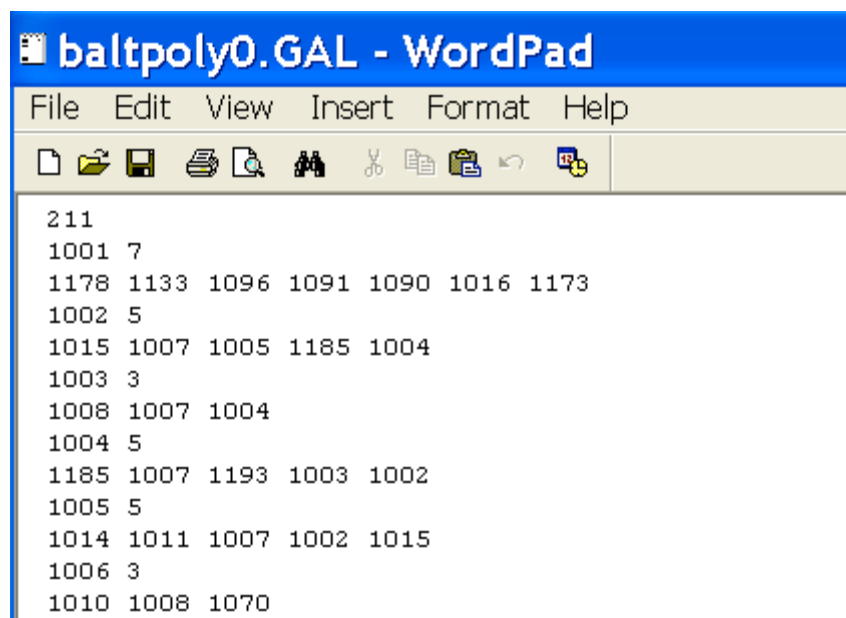


Figure 4. Old-style GAL contiguity file using ID variable.

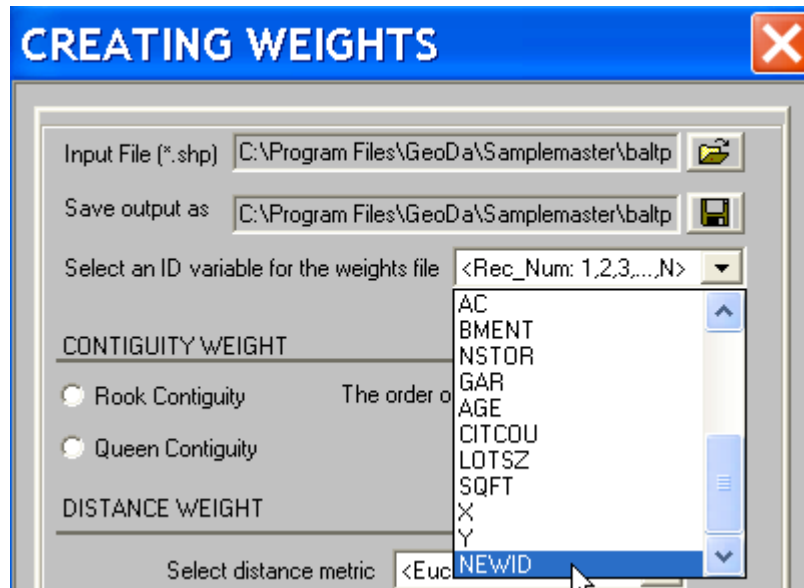


Figure 5. Creating Rook contiguity weights and specifying an ID variable.

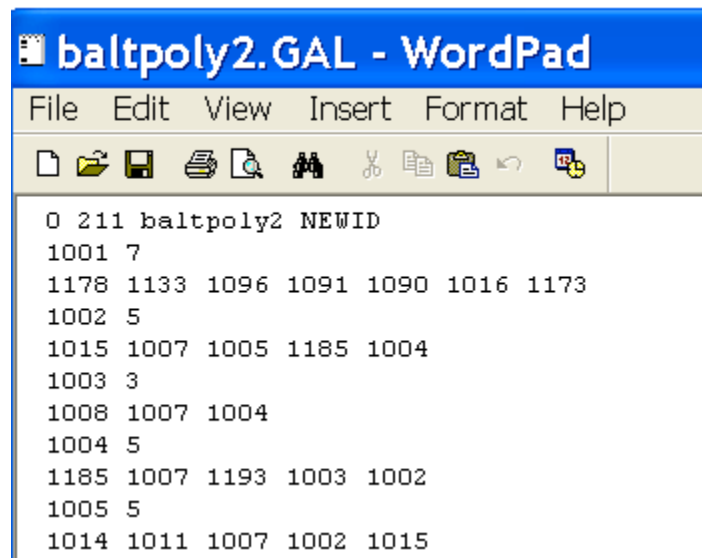


Figure 6. Contiguity file with new format header line.

You invoke `read.gal2` essentially in the same way as `read.gal`, with the important distinction that only the GAL file can be used as input (only one parameter is allowed). For example, using `baltpoly0.GAL` as input, followed by the `summary` command, yields:

```

> baltnb22 <- read.gal2("baltpoly0.GAL")
> summary(baltnb22)
Connectivity of baltnb22 with the following attributes:
List of 5

```



```

$ class      : chr "nb"
$ region.id: chr [1:211] "1001" "1002" "1003" "1004" ...
$ gal       : logi TRUE
$ call      : logi TRUE
$ sym       : logi TRUE
NULL
Number of regions: 211
Number of nonzero links: 1198
Percentage nonzero weights: 2.690865
Average number of links: 5.677725
Link number distribution:

 3  4  5  6  7  8  9 11
7 21 66 75 29  9  3  1
7 least connected regions:
1003 1006 1048 1097 1144 1157 1199 with 3 links
1 most connected region:
1208 with 11 links

```

Note how this is identical to the previous result. There is one important restriction to the `read.gal2` function as it is currently implemented. The order of the contiguity information must be the same as the order of the records in the data frame. Again, if both are generated from the same shape file using GeoDa, this should be fine. However, it is not a good assumption to use in general and it will create problems when there is no one to one match between the order of observations in the data frame that holds the `ID` variable and the order in which the contiguity information is entered in the GAL file. In `read.gal2`, the ids are constructed from the contents of the GAL file, but they are not checked against the values of the ID variable in the data frame. When both the data frame and the GAL file are created by GeoDa, this should be fine, but in other circumstances, this may create errors.

Converting GAL files to neighbor objects – New Format

The new format GAL files created by GeoDa contain a slightly more elaborate header line, as in Figure 6, with four items on the first line: 0, a place holder; 211, the number of observations; `baltpoly2`, the name of the shape file from which the contiguity information was derived; and `NEWID`, the ID variable. The third function contained in the `read.gal3.R` helper file is `read.gal3`, which is the safest of the three. In `read.gal3`, the ID variable name is read from the GAL file header line and matched to a variable in the currently attached data frame. The values in the GAL file are matched to the observation sequence in the data frame by checking the ID variable directly, and any mismatches are detected as errors. As long as the ID variable is in fact present in the current work space, this will guarantee a correct result, irrespective of the order in which the contiguity information is stored in the GAL file. In addition, “islands” are dealt with correctly and don’t generate error messages.

If the ID variable is not present in the current work space (or if there is a typo; note that variable names are case sensitive) an error message results:

```
> test <- read.gal3("baltpolybad.GAL")
```

```
Error in get(x, envir, mode, inherits) : variable "NEWID1" was not found
```

To cause the error, a new GAL file was created (`baltpolybad.GAL`) by changing the ID variable name to `NEWID1`. The current data frame does not contain this variable, hence the error message.

The `read.gal3` function is invoked in the same manner as `read.gal2` and only the file name for the GAL file must be specified. No other parameters can be included. For example, use the `baltpoly2.GAL` file from Figure 6, and invoke `read.gal3` followed by `summary`. This yields, as before:

```
> baltnb2 <- read.gal3("baltpoly2.GAL")
> summary(baltnb2)
Connectivity of baltnb2 with the following attributes:
List of 5
 $ class      : chr "nb"
 $ region.id  : chr [1:211] "1001" "1002" "1003" "1004" ...
 $ gal        : logi TRUE
 $ call       : logi TRUE
 $ sym        : logi TRUE
NULL
Number of regions: 211
Number of nonzero links: 1198
Percentage nonzero weights: 2.690865
Average number of links: 5.677725
Link number distribution:

 3  4  5  6  7  8  9 11
 7 21 66 75 29  9  3  1
7 least connected regions:
1003 1006 1048 1097 1144 1157 1199 with 3 links
1 most connected region:
1208 with 11 links
```

Characteristics of spatial weights.

The characteristics of spatial weights are obtained with the `summary` function applied to a *neighbor object*, as illustrated in the examples given above. The same result is also obtained by applying the `summary.nb` function, as in

```
> summary.nb(baltnb2)
```

This yields a brief description of the number of observations, the “density” of the spatial weights matrix (percentage nonzero weights), the average number of neighbors (links) per observation and the frequency distribution of the neighbor count. It also lists the ids of the least and most connected observations, as well as any islands, if present. Islands must be dealt with, either by manually editing the neighbor list to “connect” them to some other observation or by eliminating them from the analysis. This can be accomplished by means of the `edit.nb` command (see the `spdep` package manual, p. 20, for technical details).

It is important to note that islands *never* count in an analysis of spatial autocorrelation, since only connected observations are included in the computation of the statistics.

The `plot` command can also be used to obtain a descriptive overview of the connectedness structure in a neighbor object. It can be applied generically to a neighbor object, as `plot`, or explicitly as `plot.nb`. The name of the neighbor object and a matrix of X, Y coordinates must be supplied (note the use of `cbind` to construct the matrix from the data frame variables), as well as any of the other usual R graphics parameters. For example, applied to `baltnb2`, this yields the plot shown in Figure 7 (with the titles added for extra effect):

```
> plot.nb(baltnb2, cbind(X, Y), col="red")
> title(main="First Order Contiguity from Thiessen Polygons")
> title(xlab="Baltimore Point Data")
```

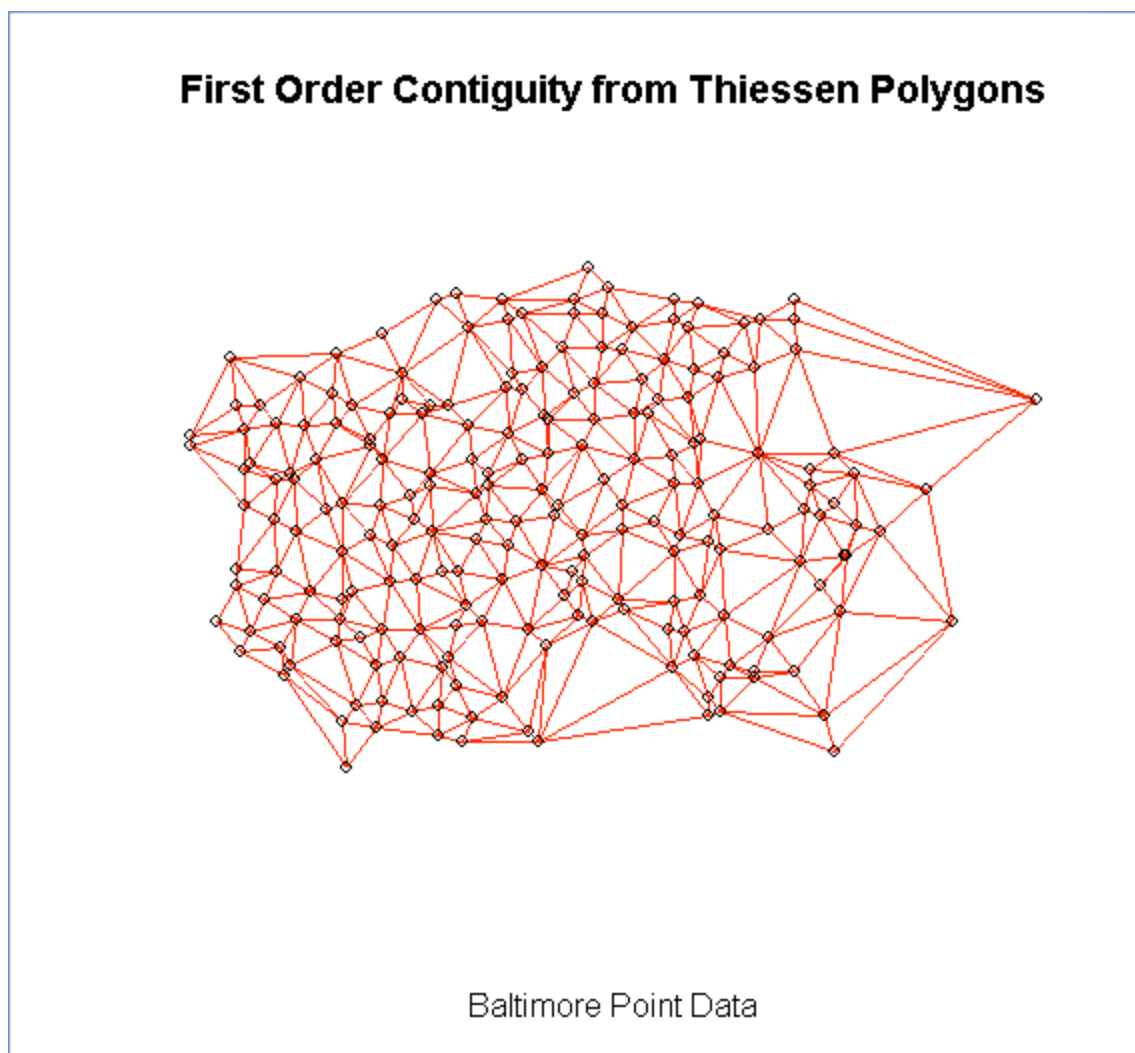


Figure 7. Plot of connectedness structure.

Spatial weights objects in spdep

The neighbor list (`nb`) is only one of several classes that handle spatial contiguity information in `spdep`. For example, spatial weights objects (`listw`) also contain actual values for the weights and are used in the spatial autocorrelation tests and the estimation of spatial regression models. Spatial weights can be constructed for a range of spatial objects, such as points, grid cells, graphs, polylists (polygon lists), as well as the GAL files discussed earlier (consult the `spdep` package description for technical details). The basic logic in the functions that handle construction and transformation between the various classes boils down to:

spatial object \rightarrow neighbor list (`nb`) \rightarrow spatial weights (`listw`)

For example, the functions `cell2nb`, `graph2nb` and `knn2nb` create neighbor lists from, respectively, grid cells, graph structures and k-nearest neighbor objects. K-nearest neighbor objects themselves are constructed from point coordinates using the `knearneigh` function. The function `dnearneigh` creates a neighbor list based on a distance cut off criterion applied to point data. The conversion of the neighbor lists to spatial weights is implemented with the `nb2listw` function.

To examine the difference between a neighbor list and a spatial weights object more closely, first consider the output of a `print` function applied to `baltnb2`:

```
> print(baltnb2)
[[1]]
[1] 16 90 91 96 133 173 178

[[2]]
[1] 4 5 7 15 185

[[3]]
[1] 4 7 8
...
...
[[211]]
[1] 9 68 76 209 210

attr(,"class")
[1] "nb"
attr(,"region.id")
 [1] "1001" "1002" "1003" "1004" "1005" "1006" "1007" "1008" "1009"
[11] "1011" "1012" "1013" "1014" "1015" "1016" "1017" "1018" "1019"
[211] "1201" "1202" "1203" "1204" "1205" "1206" "1207" "1208" "1209"
[211] "1211"
attr(,"gal")
```

```
[1] TRUE
attr(,"call")
[1] TRUE
attr(,"sym")
[1] TRUE
```

The first set of results show, for each observation, the list of neighbor ids. Then follows the class type and a set of labels for the observations (`region.id`). At the end of the listing, a series of characteristics or “attributes” are shown, such as how the neighbor list originated (`gal`) and whether or not it is symmetric (`sym`).

You convert a neighbor list object to a spatial weights object with the `nb2listw` function, as in:

```
> baltlw2 <- nb2listw(baltnb2)
```

Note that several options are available, such as the type of weights standardization (the default, “W” is row-standardization), the explicit specification of weights (the default is that they are computed from the neighbor information) and how to deal with islands (the default is to generate an error function). Technical details can be found in the `spdep` package manual (pp. 60-62). The new object created by `nb2listw` is no longer a neighbor list, which means that the `summary` function is no longer defined:

```
> summary(baltlw2)
Error in summary.nb(baltlw2) : Not a neighbours list
In addition: Warning message:
the condition has length > 1 and only the first element will be used
in: if (class(nb) != "nb") stop("Not a neighbours list")
```

The contents of the new object can be found from a `print` command:

```
> print(baltlw2)
$style
[1] "W"

$neighbours
[[1]]
[1] 16 90 91 96 133 173 178

[[2]]
[1] 4 5 7 15 185

...
...
[[211]]
[1] 9 68 76 209 210

attr(,"class")
[1] "nb"
attr(,"region.id")
[1] "1001" "1002" "1003" "1004" "1005" "1006" "1007" "1008" "1009"
"1010"
```

```

...
...
[211] "1211"
attr(,"gal")
[1] TRUE
attr(,"call")
[1] TRUE
attr(,"sym")
[1] TRUE

$weights
$weights[[1]]
[1] 0.1428571 0.1428571 0.1428571 0.1428571 0.1428571 0.1428571
0.1428571

$weights[[2]]
[1] 0.2 0.2 0.2 0.2 0.2

...
...
$weights[[211]]
[1] 0.2 0.2 0.2 0.2 0.2

attr(,"binary")
[1] TRUE
attr(,"W")
[1] TRUE

attr(,"class")
[1] "listw" "nb"
attr(,"region.id")
[1] "1001" "1002" "1003" "1004" "1005" "1006" "1007" "1008" "1009"
"1010"
...
...
[211] "1211"
attr(,"call")
nb2listw(neighbours = baltnb2)

```

The most interesting part in this listing is the section listing the `$weights`, which contains a list of the actual spatial weights values to be used in further analysis, such as the computation of spatial autocorrelation statistics.

K-nearest neighbor weights

When a data frame contains the X, Y coordinates of point locations (or of the polygon centroids), spatial weights can be constructed that contain the k (any small positive integer) nearest neighbors for each observation. In `spdep`, this is a two-step process. First, an object of class `knn` must be constructed from the location coordinates, using the function `knearneigh`. Second, this object must be converted to a neighbor list, using the function `knn2nb`.

The `knearneigh` function requires a matrix with the X, Y coordinates (typically extracted from a data frame using the `cbind` command) and a value for the number of nearest

neighbors. The default is $k = 1$, which only gives *the* nearest neighbor. For example, using the X, Y coordinates for the Baltimore points and setting $k = 4$, a `knn` object is constructed as given below. The output of the `summary` command is not very informative, basically listing the dimensions of lists and matrices. The `print` command reveals that the `knn` object contains a list of k neighbor ids (observation numbers) for each observation (`nn`), as well as the original coordinates (`x`), in addition to some descriptive information (the number of points, `np`, the value for k , the `class` and the `call`).

```
> baltknn <- knearneigh(cbind(X,Y),k=4)
> summary(baltknn)
      Length Class  Mode
nn      844   -none- numeric
np        1   -none- numeric
k         1   -none- numeric
dimension  1   -none- numeric
x       422   -none- numeric
> print(baltknn)
$nn
      [,1] [,2] [,3] [,4]
[1,]   96   16   90  133
[2,]    5    4    7  185
[3,]    4    7    2    5
[4,]    2    3    7    5
...
...
[210,]   76  209   75  211
[211,]   68  209    9   65

$np
[1] 211

$k
[1] 4

$dimension
[1] 2

$x
      X      Y
[1,] 907.0 534.0
[2,] 922.0 574.0
[3,] 920.0 581.0
...
...
[210,] 919.0 554.0
[211,] 914.0 558.0

attr(,"class")
[1] "knn"
attr(,"call")
knearneigh(x = cbind(X, Y), k = 4)
```

The `knn` object `baltknn` is passed as a parameter to the `knn2nb` function to convert it into a neighbor list. The other options are to specify a character vector of id values (the

default is to use sequence numbers) and to force symmetry (the default is not to force symmetry). Typically, a k-nearest neighbor connectedness is not a symmetric relation. Setting the option `sym = TRUE` adds connections to force symmetry. This is usually not a good idea since it changes a fundamental aspect of the concept. In the example below, the `row.names` were set to the `NEWID` variable.

```
> baltknnb <- knn2nb(baltknn, row.names=NEWID)
```

The new object is a neighbor list, and, as before, its characteristics are revealed with a `summary` command. Note that for this type of neighbor list, this is not that useful, except as a quick check for possible problems, since each observation has the same number of neighbors by construction (the list of least connected has been truncated in the listing that follows).

```
> summary(baltknnb)
Connectivity of baltknnb with the following attributes:
List of 6
 $ region.id: int [1:211] 1001 1002 1003 1004 1005 1006 1007 1008 1009
1010 ...
 $ call      : language knearneigh(x = cbind(X, Y), k = 4)
 $ sym       : logi FALSE
 $ type      : chr "knn"
 $ knn-k     : num 4
 $ class     : chr "nb"
NULL
Number of regions: 211
Number of nonzero links: 844
Percentage nonzero weights: 1.895735
Average number of links: 4
Link number distribution:

 4
211
211 least connected regions:
1001 1002 1003 1004 1005 1006 1007 1008 1009
```

Distance-based spatial weights

Another important category of spatial weights can be constructed from the distance between point locations (or polygon centroids). In `spdep`, this is implemented in the `dnearneigh` function, which takes as input a matrix of coordinates and a lower and upper distance bound (as well as, optionally, a variable containing region ids). All points that are within the defined distance band from each other are categorized as neighbors. It is important to note that the distance calculation applies only to Euclidean distance, which requires that the coordinates are *projected* (not simply latitude, longitude).

A slight complication is that the value for the lower and upper distance bound must be specified beforehand. Typically the lower bound is zero, but the upper distance bound should be such that no observations become islands. In `GeoDa`, this is shown in the weights construction dialog, but in `spdep`, this takes a little more work.

The workaround is to start with a first order nearest neighbor list, using the `knearneigh` and `knn2nb` functions discussed above. Next, the function `nbdists` is used to obtain the Euclidean distances between the neighbors. By construction, this is the nearest neighbor distance. Finally, the maximum of the nearest neighbor distances is extracted and used as the input for the upper bound in the `dnearneigh` function. This guarantees that each observation has at least one neighbor, but it often results in an undesirable overall connectedness structure when the inter-point distances are very heterogeneous (or, when the polygons vary considerably in size).

To illustrate this procedure, start by constructing the first order nearest neighbors for the Baltimore point data, as:

```
> baltk1 <- knearneigh(cbind(X,Y))
> baltknn1 <- knn2nb(baltk1)
> summary(baltknn1)
Connectivity of baltknn1 with the following attributes:
List of 6
 $ region.id: chr [1:211] "1" "2" "3" "4" ...
 $ call      : language knearneigh(x = cbind(X, Y))
 $ sym       : logi FALSE
 $ type      : chr "knn"
 $ knn-k     : num 1
 $ class     : chr "nb"
NULL
Number of regions: 211
Number of nonzero links: 211
Percentage nonzero weights: 0.4739336
Average number of links: 1
Link number distribution:

 1
211
```

Note that the `k =` option is not necessary, since `k = 1` is the default. Also, this time, no `row.names` were specified. The `baltknn1` object is a neighbor list with one neighbor for each observation. The list with nearest neighbor distances is obtained from `nbdists`, with `baltknn1` and the matrix of coordinates as input:

```
> baltdist1 <- nbdists(baltknn1,cbind(X,Y))
> baltdist1
[[1]]
[1] 5.09902

[[2]]
[1] 4

[[3]]
[1] 4.242641
...
...
[[211]]
[1] 0.8602325
```

```
attr("class")
[1] "nbdist"
attr("call")
nbdists(nb = baltknn1, coords = cbind(X, Y))
```

The result, `baltdist1`, is a list object. The `max` function is not defined for lists, but it is defined for vectors. The list can be converted to a vector object by means of the `unlist` function:

```
> baltdistvec <- unlist(baltdist1)
> baltdistvec
[1] 5.0990195 4.0000000 4.2426407 4.1231056 2.0000000 3.1622777
[7] 2.0000000 3.1622777 3.3541020 3.1622777 4.2426407 2.2360680
```

Now, the `max` function applied to the vector `baltdistvec` will yield the greatest nearest neighbor distance, such that no observation becomes an island:

```
> baltmaxd <- max(baltdistvec)
> baltmaxd
[1] 21.31901
```

At this point, you have all the pieces necessary to construct a distance-based spatial weight using `dnearneigh`: pass the matrix of coordinates, the lower bound (0), and upper bound (`baltmaxd`) and, optionally, a variable with identifiers (`NEWID`), as

```
> baltdnb <- dnearneigh(cbind(X, Y), 0, baltmaxd, row.names=NEWID)
```

Apply the `summary` function to the new object and note that all is not well. The procedure has truncated the maximum distance and left one observation (`ID = 1102`) as an island, with no links. The value used in the computation is 21.3. You can fix this by manually entering 21.32, as illustrated next.

```
> summary(baltdnb)
Connectivity of baltdnb with the following attributes:
List of 7
 $ class      : chr "nb"
 $ nbtype     : chr "distance"
 $ distances: num [1:2] 0.0 21.3
 $ region.id: int [1:211] 1001 1002 1003 1004 1005 1006 1007 1008 1009
1010 ...
 $ call       : language dnearneigh(x = cbind(X, Y), d1 = 0, d2 =
baltmaxd, row.names = NEWID)
 $ dnn        : num [1:2] 0.0 21.3
 $ sym        : logi TRUE
NULL
Number of regions: 211
Number of nonzero links: 7864
Percentage nonzero weights: 17.66357
Average number of links: 37.27014
Link number distribution:
```

```

 0  7 11 12 13 15 16 17 18 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36
 1  1  1  2  1  1  2  2  8  3  3  3  6  5  2  5  8  3  6  4  5  6  7  3
5  3
37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
 3  4  3  7  8  2 10  6  7  6  4  6  5  6  8  6  8  6  6  4
1 region with no links:
1102
1 least connected region:
1115 with 7 links
4 most connected regions:
1040 1075 1094 1133 with 56 links

> baltdnb1 <- dnearneigh(cbind(X,Y),0,21.32,row.names=NEWID)
> summary(baltdnb1)
Connectivity of baltdnb1 with the following attributes:
List of 7
 $ class      : chr "nb"
 $ nbtype     : chr "distance"
 $ distances: num [1:2]  0.0 21.3
 $ region.id: int [1:211] 1001 1002 1003 1004 1005 1006 1007 1008 1009
1010 ...
 $ call       : language dnearneigh(x = cbind(X, Y), d1 = 0, d2 = 21.32,
row.names = NEWID)
 $ dnn        : num [1:2]  0.0 21.3
 $ sym        : logi TRUE
NULL
Number of regions: 211
Number of nonzero links: 7874
Percentage nonzero weights: 17.68604
Average number of links: 37.31754
Link number distribution:

 1  7 11 12 13 15 16 17 18 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36
 1  1  1  1  2  1  2  2  8  3  3  3  5  6  2  5  8  3  6  3  6  6  7  1
7  3
37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
 2  5  3  6  9  2 10  6  7  6  4  6  5  6  7  7  8  6  6  3  1
1 least connected region:
1102 with 1 link
1 most connected region:
1075 with 57 links

```

The `summary` illustrates a common problem with spatial weights derived from distance bands. The distribution of the number of neighbors is highly irregular, including both very small numbers (1, 7 neighbors) as well as very large numbers (56, 57) for observations that are more densely distributed in space. The use of this weights matrix in spatial regression is likely to violate some of the fundamental regularity conditions required to obtain the asymptotic properties of various estimators and test statistics. The application of a summary function to any constructed spatial weights object will indicate when such trouble may be expected. *It should be standard practice!*