areal1

Philip Dixon

3/6/2020

areal1.r:

```
library(sp)
library(spdep)  # most functions for areal data analysis
```

```
## Warning: package 'spdep' was built under R version 3.6.2

## Loading required package: spData

## Warning: package 'spData' was built under R version 3.6.2

## To access larger datasets in this package, install the spDataLarge
## package with: `install.packages('spDataLarge',
## repos='https://nowosad.github.io/drat/', type='source')`

## Loading required package: sf

## Linking to GEOS 3.6.1, GDAL 2.2.3, PROJ 4.9.3
```

```
library(rgdal)  # functions to read ARC shape files
```

```
## rgdal: version: 1.4-3, (SVN revision 828)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.3, released 2017/11/20
##  Path to GDAL shared files: C:/Users/pdixon/Documents/R/win-library/3.6/rgdal/gdal
##  GDAL binary built with GEOS: TRUE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: C:/Users/pdixon/Documents/R/win-library/3.6/rgdal/proj
##  Linking to sp version: 1.3-1
```

```
library(spData) # data sets for the sp package, includes the Auckland shapefile
```

functions for plotting areal data, working with weights, and computing / testing spatial correlation

Example is spdiv.txt. contains species diversity in 20cm x 20cm subquadrats. The data file is organized as a row x column matrix reflecting the plot layout no header (column names) and no row names. To use as a spatial object, need data as one row per obs, with x and y coords. Here's how to read that sort of data.

scan() is a very basic way to read data. It reads all values in a file by reading all values on one line in sequence, then reading all values on the second line, etc. through the file. The result is a vector. I use expand.grid to create all combinations of x values (x = 1, 2, .. 8) and y values (y = 1, 2, .. 8). Then combine both pieces into one data frame. Temp is a data frame, so the variable names get reused, and I provide a name for the response variable. spdiv= names the data frame column. the contents are the right hand side of = (spdivY).

```
spdivY <- scan('spdiv.txt')
temp <- expand.grid(x=1:8, y=1:8)
spdiv.sp <- data.frame(temp, spdiv = spdivY)
head(spdiv.sp)
```

```
##   x y spdiv
## 1 1 1     3
## 2 2 1     5
## 3 3 1     7
## 4 4 1     8
## 5 5 1     9
## 6 6 1     6
```

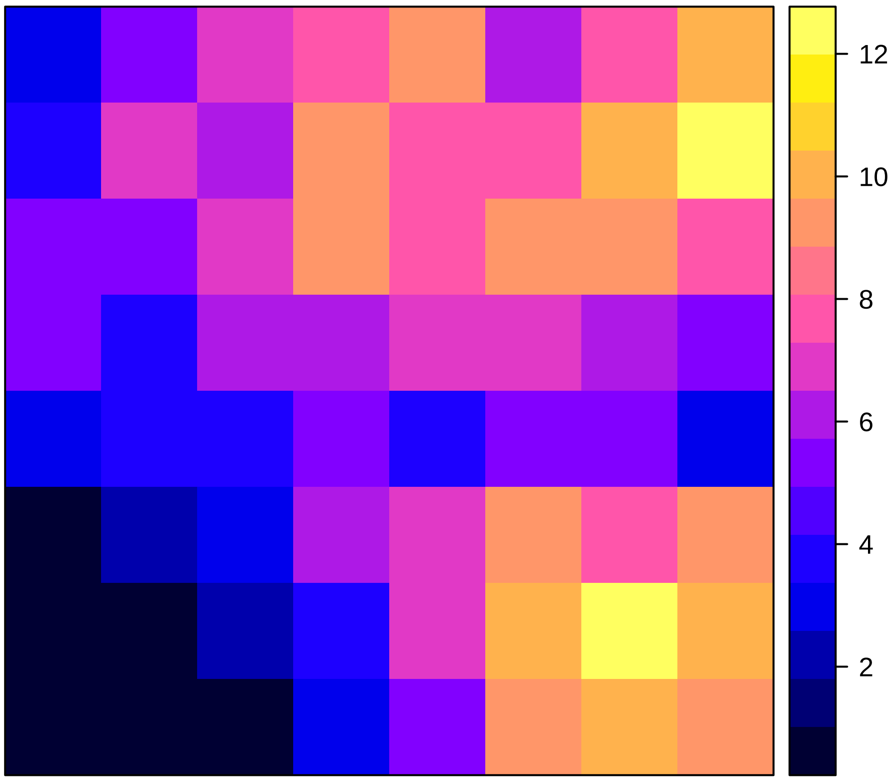
One detail: (can be ignored if you want to): The top row is numbered y=1, so it will plot at the bottom You can switch so top row is numbered y = 8 so it plots at the top.

```
spdiv.sp$y <- 9 - spdiv.sp$y
head(spdiv.sp)
```

```
##   x y spdiv
## 1 1 8     3
## 2 2 8     5
## 3 3 8     7
## 4 4 8     8
## 5 5 8     9
## 6 6 8     6
```

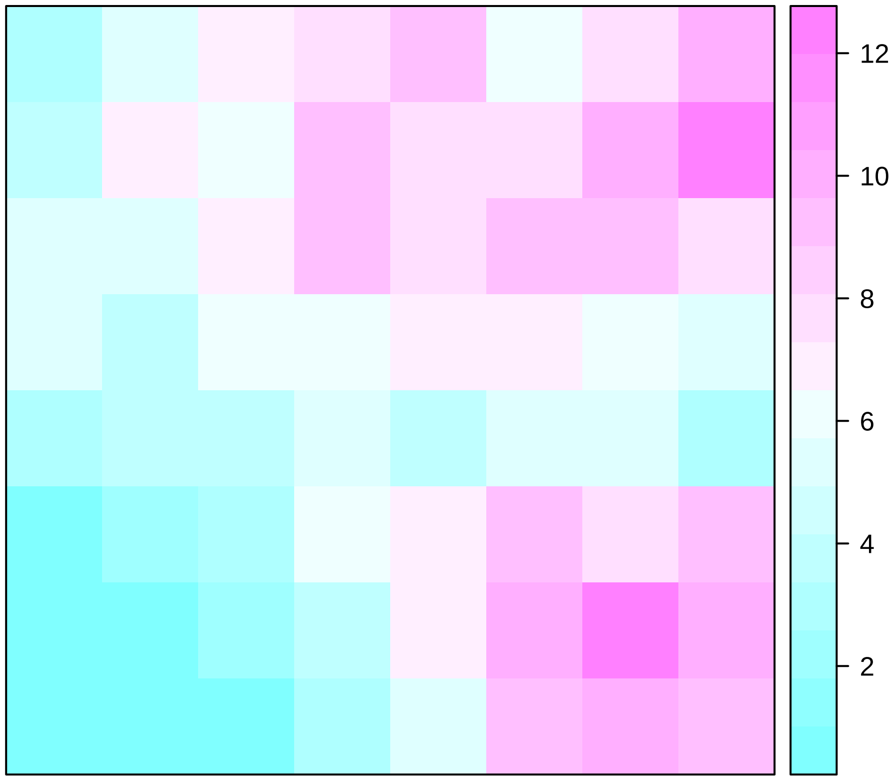
Convert to a gridded spatial object in the usual way, and look at the data.

```
coordinates(spdiv.sp) <- c('x','y')
gridded(spdiv.sp) <- T
spplot(spdiv.sp,'spdiv')
```



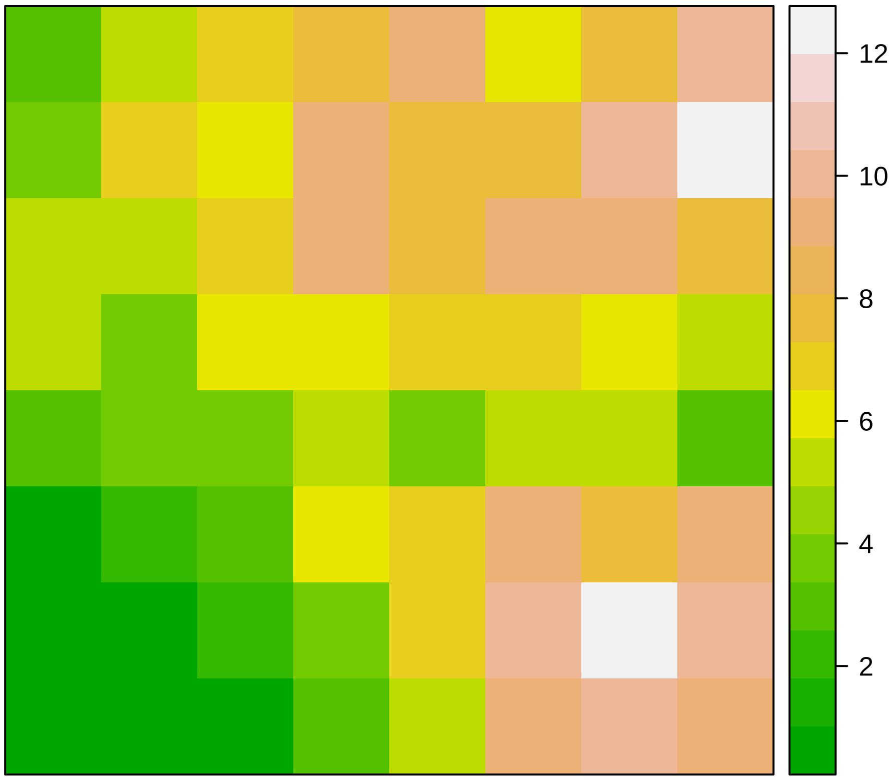
Nicer color schemes. spplot() uses 16 colors by default, so specify a palette with 16 levels. That's the number in the parentheses. ?cm.colors will list all the built-in palettes.

```r
spplot(spdiv.sp,'spdiv', col.regions=cm.colors(16))
```



or:

```r
spplot(spdiv.sp,'spdiv', col.regions=terrain.colors(16))
```



Constructing neighbor lists.

You need a weight matrix for almost all areal data analyses.
The basic approach has two steps:

- identify neighbors for each region
- use neighbors to define weights

These can sometimes to be combined into a single step.

In general, spdep tries to not store the weights as a matrix because that is very inefficient (lots of 0's). It prefers to store neighbors as a 'neighbor list' and to store weights as a 'weight list'. There are quite a few ways to generate neighbor information and convert it into other forms.

- dnearneigh uses distance range to identify neighbors
- knearneigh uses closest k (that you specify) neighbors
- nb2listw converts a neighbor list to a weight list. Weights can be computed in multiple ways.
- mat2listw converts a square matrix of weights to the weight list form
- poly2nb() converts ARC shape file polygons into neighbor information. An example of this is below (Auckland data).

The next few examples all use the spdiv data.

Rook's neighbors: The coordinates are 1, 2, ... so the rook's neighbors of a focal area are those areas that are 1 unit apart. dnearneigh(data, min, max) uses a distance range to define neighbors. The arguments are the spatial data with coordinates and the desired range of distances, specified as min (bottom of the range) and max (top of the range). So rook's neighbors are min=0, max=1. If the grid is scaled differently, you'll have to change the limits. E.g., if the coordinates are 0, 10, 20, … then you want min=0, max=10.

Queen's neighbors: These include the diagonals, which are sqrt(2) from the focal area. min=0, max=sqrt(2).

If you print the first few elements of the neighbor list, you see the structure. This is a list with one component for each area. That component is the location numbers of the neighbors, stored as a vector. Location 1 has two rook's neighbors (areas 2 and 9, i.e. the area to the right and the below). That information is sd.nb1[[1]].

```
sd.nb1 <- dnearneigh(spdiv.sp, 0, 1)        # Rooks neighbors
sd.nb2 <- dnearneigh(spdiv.sp, 0, sqrt(2))  # Queens neighbors

head(sd.nb1)
```

```
## [[1]]
## [1] 2 9
##
## [[2]]
## [1]  1  3 10
##
## [[3]]
## [1]  2  4 11
##
## [[4]]
## [1]  3  5 12
##
## [[5]]
## [1]  4  6 13
##
## [[6]]
## [1]  5  7 14
```

To compute the number of neighbors an area, look at the length of the appropriate neighbor vector. You can repeat that for all areas in the list using sapply, which repeats a function for each element of a list and returns the result as a vector. (lapply is similar but returns the result as a list).

```
length(sd.nb1[[1]])
```

```
## [1] 2
```

```
nneigh <- sapply(sd.nb1,length)
table(nneigh)
```

```
## nneigh
##  2  3  4
##  4 24 36
```

For rooks neighbors, 4 areas have 2 neighbors, 24 have 3 and 36 have 4.

Constructing weight lists

Again, to save space, weight information is stored as a list, with neighbor ids and associated weights.

- nb2listw() creates the weight list. The style= argument specifies what type of weight. style='B' is binary (0/1) weights. style='W' is row-standardized (sum of weights for each area = 1). There are other options but I have no experience with them.
- listw2mat() converts a weight list to a matrix (but don't print the whole thing)

```r
sd.w1 <- nb2listw(sd.nb1, style='B')   # 0/1 weights
sd.w2 <- nb2listw(sd.nb1, style='W')   # row standardized weights (each row sums to 1)

temp <- listw2mat(sd.w1)
temp[1:10,1:10]          # prints only 1st 10 rows and 10 columns
```

```
##    [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## 1     0    1    0    0    0    0    0    0    1     0
## 2     1    0    1    0    0    0    0    0    0     1
## 3     0    1    0    1    0    0    0    0    0     0
## 4     0    0    1    0    1    0    0    0    0     0
## 5     0    0    0    1    0    1    0    0    0     0
## 6     0    0    0    0    1    0    1    0    0     0
## 7     0    0    0    0    0    1    0    1    0     0
## 8     0    0    0    0    0    0    1    0    0     0
## 9     1    0    0    0    0    0    0    0    0     1
## 10    0    1    0    0    0    0    0    0    1     0
```

```r
temp <- listw2mat(sd.w2)
temp[1:5,1:5]
```

```
##        [,1]      [,2]      [,3]      [,4]      [,5]
## 1 0.0000000 0.5000000 0.0000000 0.0000000 0.0000000
## 2 0.3333333 0.0000000 0.3333333 0.0000000 0.0000000
## 3 0.0000000 0.3333333 0.0000000 0.3333333 0.0000000
## 4 0.0000000 0.0000000 0.3333333 0.0000000 0.3333333
## 5 0.0000000 0.0000000 0.0000000 0.3333333 0.0000000
```

Don't use head() to look at a weight list, unless you want a lot of output. The weight list is a list of lists, one is the neighbor list, another is a list of weights. head() will print the first 6 elements of the top list, which includes everything in the weight list.

reading ARC shape files and creating neighbor lists from them

The Auckland shape file is distributed as part of the spData package. In general, to access datasets distributed in packages, you need to first use data(something) to access the "something" data set. If the "something" is a data frame, that's all you need to do. Auckland is an ARC shape file, so we need to read it from the system folder for the spData package. system.file() fills in the full name of the file. You'll sometimes see [1] at the end of the system.file(). Usually system.file() returns one file name and [1] is irrelevant (you're requesting the first element of a 1 element vector). If, for some reason, system.file() returns two (or more) file names, the [1] picks out on the first one.

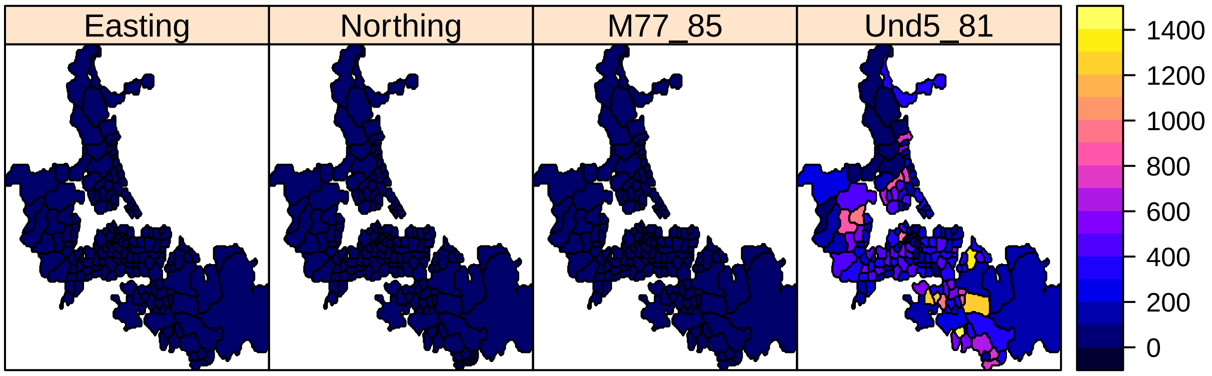
The information for an ARC shape file is contained in multiple files (If you look in the folder, you'll see 3 Auckland files). If you copy a shapefile from one computer to another, you will need all three files to get the shapefile and its data. The readOGR() function reads the shapefile and all the associated information. Since Auckland includes data, the result is a SpatialPolygonsDataFrame object.

The poly2nb() function creates a neighbor list from the polygons. A neighbor can be defined two ways: sharing a single point (or nearly the same point) or sharing more than one point. The first is like queens neighbors; the second is like rooks neighbors. This choice is specified by queen=. The default is T, so only need a single point. Sometimes polygons are meant to be adjacent but there is tiny gap. Specifying snap= means that any distance less than snap apart is contiguous.

```r
data(auckland)  # access a system data file
auckland <- readOGR(system.file("shapes/auckland.shp", package="spData"))
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\pdixon\Documents\R\win-library\3.6\spData\shapes\auckland.shp", layer: "auckland"
## with 167 features
## It has 4 fields
```

```
spplot(auckland)
```



```
auckland.nb <- poly2nb(auckland)  # default polygons (queen=T, must overlap)
auckland.nb2 <- poly2nb(auckland, queen=F, snap=1)
#rooks neighbors with < 1m apart considered contiguous
# distance is in meters because coordinates are Northing and Easting

# in case you want to see what's going on:
system.file("shapes/auckland.shp", package="spData")
```

```
## [1] "C:/Users/pdixon/Documents/R/win-library/3.6/spData/shapes/auckland.shp"
```

```
list.files(system.file("shapes/auckland.shp", package="spData"))
```

```
## character(0)
```

Bivand describes interfaces to the GRASS public domain GIS and ways to work with GoogleMaps and GoogleEarth files.

If polygon information and the data are in a different files (and not part of a ARC shape file complex), read that file into a data frame, then merge the data with the SpatialPolygons by:

auckland2 <- SpatialPolygonsDataFrame(auckland, data, match.ID='id')

The first argument is the name of the SpatialPolygons object, data is the name of the data frame, and 'id' is the name of a variable to match data frame rows to polygon names. Each polygon has a name. See head(row.names(auckland)) to see the names of the first few. id is variable in the data frame that identifyies the polygon for that row of data.

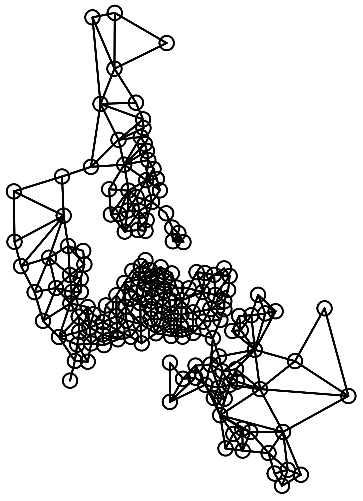
This matching is usually the trickiest part of merging. But, it is crucial, so that data is attached to the appropriate polygon. If necessary, rows in the data frame will be reordered so they match the appropriate polygons.

If you are absolutely sure that the first polygon matches the first row, in the data frame, the second matches the second, . . . , then specifying match.ID=F, i.e. auckland2 <- SpatialPolygonsDataFrame(auckland, data, match.ID=F) suppresses checking names and reordering if necessary.

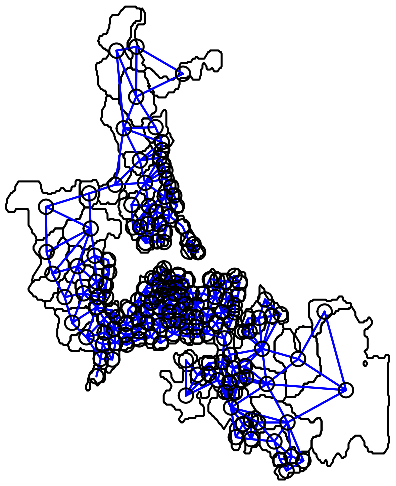
Adding neighbor links to a plot

It can be very useful to plot a set of polygons and add line segments connecting neighbors. Here are two ways to do that. The first just draws the connections. The second overlays the connections on the polygons.

```
plot(auckland.nb, coordinates(auckland))
```



```
plot(auckland)
plot(auckland.nb, coordinates(auckland), add=T, col=4)
```



Both require the neighbor list and the centroids of the polygons. coordinates() returns the centroid of each polygon. To add the neighbor connections to the polygons, you need to plot the polygons first, the add (note add=T) the connections. col=4 plots connections in blue (color # 4).

Editing neighbor lists (optional), does not work in R Studio

edit.nb() provides a graphical editor for links. This will plot the links tnen expect input from the mouse. Click on the ends of an existing link and you'll be asked if you want to delete it. Click on two currently unlinked regions, and you'll be asked if you want to add that link.

If you have polygons, you can plot them in the background (Auckland example). If you don't have polygons, you need to provide the coordinates (spdiv example). Both are commented out because they don't work with RStudio. Note: You must store results, otherwise any changes are ignored.

```
#auckland.nb2 <- edit.nb(auckland.nb, poly=auckland)

#spdiv.nb1b <- edit.nb(sd.nb1, coordinates(spdiv.sp))
```

Calculating Moran's I and local Moran's I

All the spatial correlation functions are in the spdep library. They have similar arguments. All need a vector with the values for each polygon followed by the weight list. You can extract the vector from a spatial object or a data frame. The functions are:

- moran.test(): Moran I, with test using the normal approximation
- moran.mc(): Moran I with test using permutation of the values, need to specify # permutations
- localmoran(): local Moran's I

- geary.test(): Geary's c, with test using the normal approximation
- geary.mc(): Geary's c, with test using permutation

All but localmoran() have nice default print functions. The output from the two .test functions includes the Z score and p-value, then estimates from the data: estimated I (or c), its expected value and its variance. The output from the two .mc functions includes the number of simulations, the observed I (or c), its rank in the permutation distribution and p-value.

localmoran() returns a matrix with 1 row for each area. The columns are the observed local I, its expectation, its variance, the Z score and p-value. To plot a column you want, attach it to the spatial object.

```
moran.test(spdiv.sp$spdiv, sd.w1)
```

```
##
##  Moran I test under randomisation
##
## data:  spdiv.sp$spdiv
## weights: sd.w1
##
## Moran I statistic standard deviate = 8.2068, p-value < 2.2e-16
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##       0.741032282      -0.015873016       0.008506209
```

```
moran.mc(spdiv.sp$spdiv, sd.w1, 9999)
```

```
##
##  Monte-Carlo simulation of Moran I
##
## data:  spdiv.sp$spdiv
## weights: sd.w1
## number of simulations + 1: 10000
##
## statistic = 0.74103, observed rank = 10000, p-value = 1e-04
## alternative hypothesis: greater
```

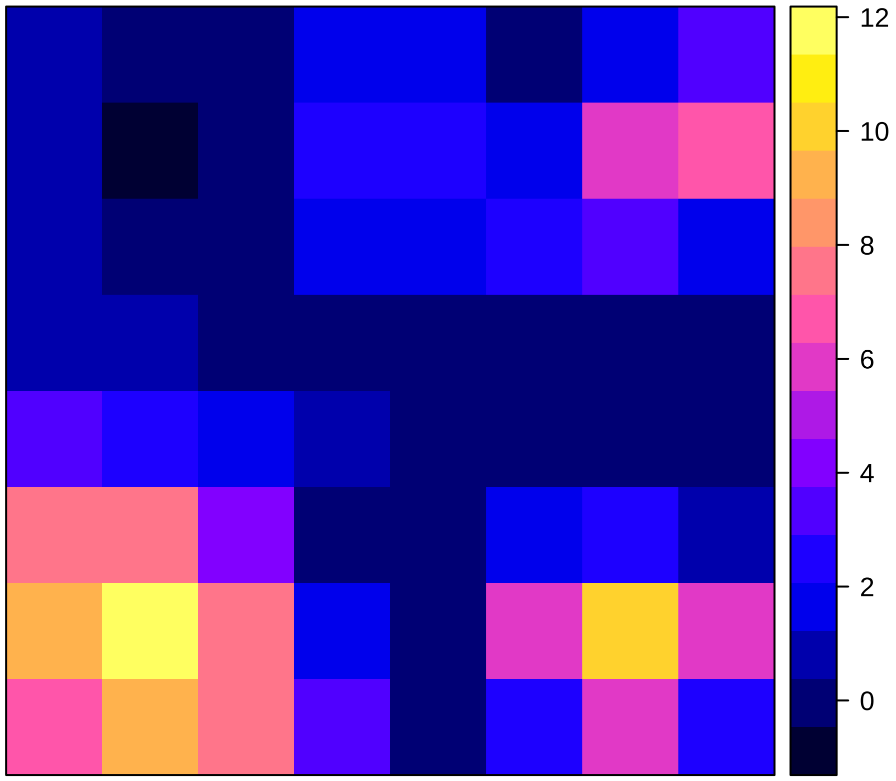
```
geary.test(spdiv.sp$spdiv, sd.w1)
```

```
##
##  Geary C test under randomisation
##
## data:  spdiv.sp$spdiv
## weights: sd.w1
##
## Geary C statistic standard deviate = 8.4923, p-value < 2.2e-16
## alternative hypothesis: Expectation greater than statistic
## sample estimates:
## Geary C statistic       Expectation          Variance
##       0.201564166       1.000000000       0.008839495
```

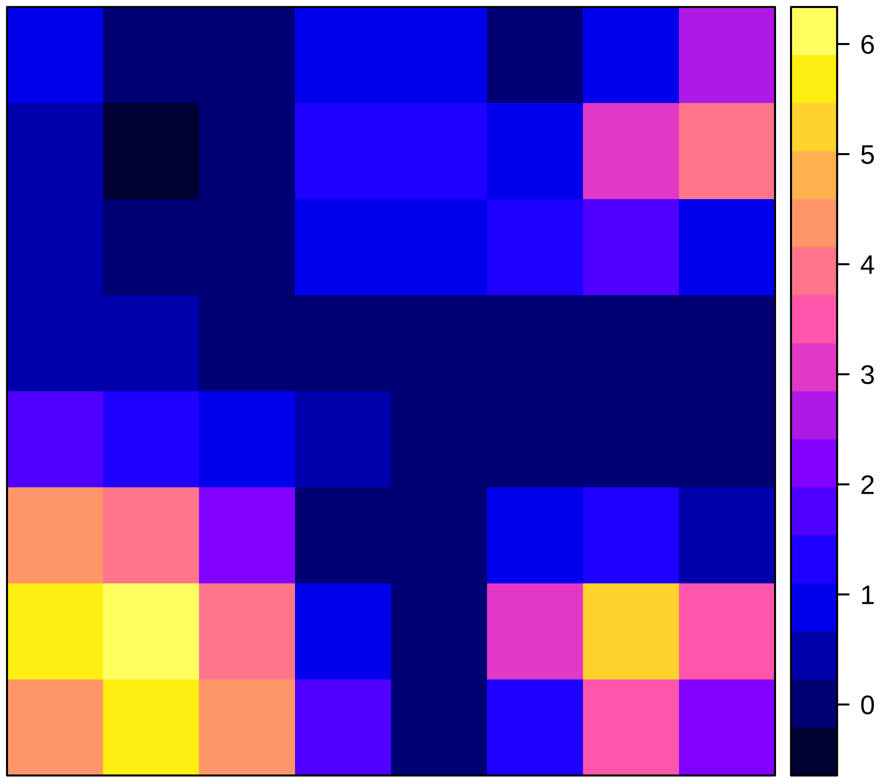
```
geary.mc(spdiv.sp$spdiv, sd.w1, 9999)
```

```
##
##  Monte-Carlo simulation of Geary C
##
## data:  spdiv.sp$spdiv
## weights: sd.w1
## number of simulations + 1: 10000
##
## statistic = 0.20156, observed rank = 1, p-value = 1e-04
## alternative hypothesis: greater
```

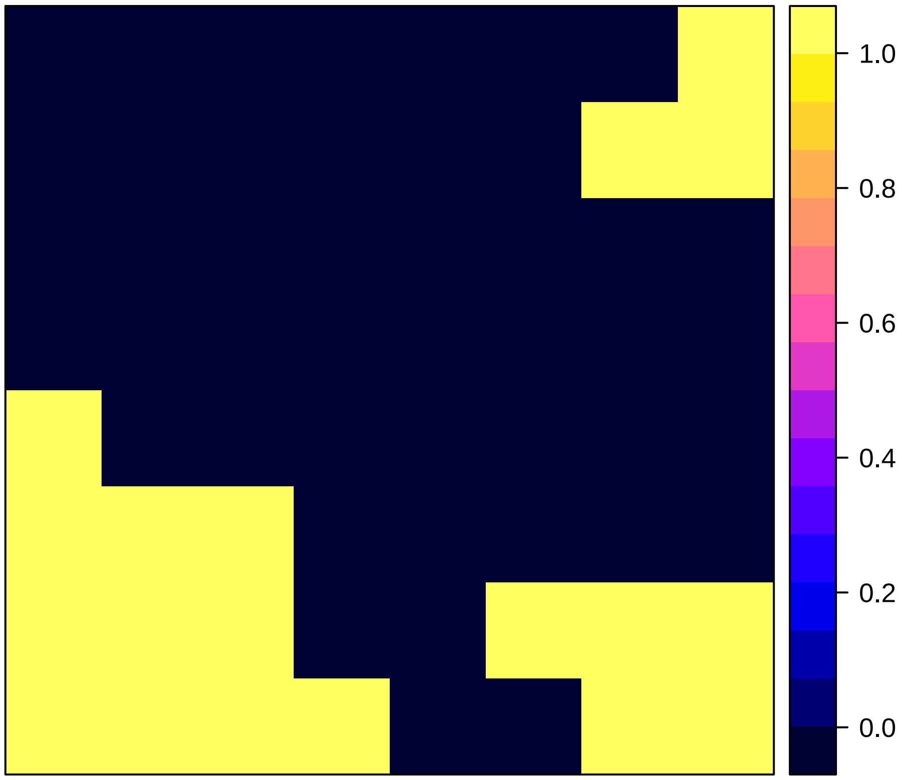
```
temp <- localmoran(spdiv.sp$spdiv, sd.w1)
spdiv.sp$I <- temp[,1]
#  extract the 1st column (I) values and put them in the spatial object
spplot(spdiv.sp, 'I')
```



```
spdiv.sp$Z <- temp[,4]
#  extract the 4th column (Z) values
spplot(spdiv.sp, 'Z')
```



```
spdiv.sp$Ip <- (temp[,5] < 0.05) + 0
#  also extract the p-value and record 1 if < 0.05 and 0 if not
#   (temp < 0.05) is a logical expression, result is T or F
#   T stored as 1, F stored as 0.  Convert to integers by adding 0
spplot(spdiv.sp, 'Ip')
```



The localmoran() function uses a normal approximation to generate p-values. This is a bit suspect because only a small data set used for each location. localmoran.sad() and localmoran.exact() compute p-values using more appropriate small-sample approximations. Bivand, p. 286 briefly discusses these. The syntax is different (an lm() model); I have no experience with either.