Spatial Regression Analysis in R
A Workbook

Luc Anselin

Spatial Analysis Laboratory
Department of Agricultural and Consumer Economics
University of Illinois, Urbana-Champaign
Urbana, IL 61801

http://sal.agecon.uiuc.edu/

Center for Spatially Integrated Social Science

http://www.csiss.org/

Preliminary Version, January 18, 2005

Contents

List of Figures

Preface

This workbook contains a set of laboratory exercises initally developed for a graduate course in *Spatial Econometrics* at the University of Illinois, as well as for the ICPSR Summer Program course on *Spatial Regression Analysis.* It consists of a series of brief tutorials and worked examples using R and its package spdep for spatial regression analysis.

Some of these materials were included in earlier tutorials available on the SAL web site. In addition, the workbook incorporates all exercise materials for the course ACE 492SE, *Spatial Econometrics*, offered during the Fall 2003 semester in the Department of Agricultural and Consumer Economics at the University of Illinois, Urbana-Champaign. There may be slight discrepancies due to changes in the version of R and spdep. In case of doubt, the latest document available on the SAL web site should always be referred to as it supersedes all previous tutorial materials.

The examples and practice exercises use the sample data sets that are available from the SAL "stuff" web site. They are listed on and can be downloaded from http://sal.agecon.uiuc.edu/data_main.php. The main purpose of these sample data is to illustrate the features of the software. Readers are strongly encouraged to use their own data sets for the practice exercises.

Warning: These notes are *not* polished, and remain very much in bullet style. They are intended to provide hints and additional pointers beyond what is available in the help files and the other tutorials. They also serve to define a sequence of tasks and techniques that can be covered during a given lab session. They are "in progress" and are still undergoing changes.

The development of this workbook has been facilitated by the continued research support through the U.S. National Science Foundation grant BCS-9978058 to the Center for Spatially Integrated Social Science (CSISS).

Exercise 1

Getting Started in R

1.1 Objectives

These notes are the first in a series to help you get started and working in R to carry out spatial regression analysis. The objective is that you work through the various examples at your own pace. This chapter and later ones will refer to the two short R tutorials available on SAL "stuff" web site, as well as to the *Introduction to R* (Venables et al. 2004), available from the CRAN site. I will refer to Venables et al. (2004) as the "Introduction" tutorial. I will refer to the other tutorials as the "Data" tutorial (*Data and spatial weights in spdep*, Anselin 2003a), and the "Regression" tutorial (*An introduction to spatial regression analysis in R,* Anselin 2003b). Further details on the spdep package can be found in Bivand and Gebhardt (2000), Bivand (2001, 2002a,b), and Bivand and Portnov (2004).

1.2 Conventions and General Advice

Some notational conventions:

- The command line will be shown as > followed by an R command.
- Command line contents will be in `monospaced` font.
- File names and file path names will be in *italics*.

A few general pointers:

- Create a separate directory for your work in R. Specify that directory as your starting directory in the R shortcut, as illustrated on p. 2 of the *Regression* tutorial.

- Know how to quit: `>q()` from the command line, or `File > Exit` using the R GUI.

- Know how to get help. One option is to invoke `>help.start()`. This launches your default browser as an interface to HTML help pages. Another option is to invoke help on a specific function or data set as `>help(name of function)`. A powerful feature is to use fuzzy search, so that you don't have to know the exact command you need (if that is what you are looking for in the first place).

- A useful way to find out more how a command works is to cut and past sample code that is contained in the help.

- The assignment symbol is `<-`. This assigns the result of what it done on the RHS to the "object" on the LHS. Often it may seem like nothing happens after you run a function. Typically that is because the result was assigned to an object without an explicit call to `print` or `summary`. After this assignment you manipulate the object, e.g., to print or summarize the results.

1.3 Sample Session

As a first orientation, work through the sample sesssion in Venables et al. (2004), Appendix A. This is a good way to get an idea of what R can do and how it works.

You can either follow along in the pdf file, or you can open the Introduction in HTML format. To do the latter, invoke the HTML help command by selecting (`>help.start()`), and click on the entry "An Introduction to R," and select "A sample session".

This set of examples requires a specific sample data set, *morley.tab.* The tutorial asks you to copy that file to your working directory. In the current version of R (2.0.1) all sample data sets are contained in the datasets package. To make it available, enter `>library(datasets)` followed by `>data(morley)`. All the variables in the morley data sets should then be available. Once you complete this, either type the commands from the sample session on the command line (to get used to the experience of unforgiven typos) or cut and paste from the HTML file. Needless to say, the latter is the preferred approach. Note that when using the morley data set from the datasets package, you need to `attach(morley)` to get direct access to the variables. At that point, you no longer need to specify `data=mm` as shown in the sample code (if you copy the commands from the html help file, this will give an error message).

After going through the sample session, you may want to experiment with some of the commands. Look up the specifics using `help( )` and cut and paste code from the examples to see how it works. You may also want to run some `demo( )` for applications (not all of them have a `demo` function implemented). For example, a nice illustration of the graphics capability is `demo(graphics)`.

Exercise 2

Getting Your Data and Weights Into R

2.1 Objectives

The purpose of this exercise is to provide guidance in turning your data and spatial weights files into usable objects in R, and to carry out some basic data and weights manipulations. The assumption is that you have the data originally as shape files and will construct the weights files in *GeoDa*. Of course, R has its own set of functions to create an analyze spatial weights, and we will consider those in a separate exercise (see also the *Data* tutorial).

You should create a new working directory for your current R project and make sure all files (data files and weights files) are copied to that directory. This is not absolutely necessary, but it's a good idea, since it avoids problems with path names, etc.

You will be using the POLICE sample data set from the SAL data archive. Make sure you download the file and unzip it into your working directory. After unzipping the file, check the `police.html` file to become familiar with the contents of the data set.

2.2 Getting Data and Weights File Ready

You first need to export the data from the shape file format to an ascii file that R can read. You also need to create some weights files using *GeoDa*. A similar exercise is described in the *Data* tutorial, using older versions of R and spdep.

2.2.1 Exporting Data with GeoDa

Load the `police.shp` file into *GeoDa*, with `FIPSNO` as the `Key Variable`. Select `Tools > Data Export > Ascii`. In the dialog, specify `police.dbf` as the input file and `police.txt` as the output file. Select all variables, except `AREA`, `CNTY_`, `CNTY_ID` and `PERIMETER`. Click on `Export` and `Done` to finish. Check the contents of the `police.txt` file and make sure it shows `82,12` on the first line and the variable names on the second line. The data, record by record, and comma-delimited make up the remainder. Copy the file to the R working directory if it is not there already.

2.2.2 Creating the Spatial Weights in GeoDa

For this exercise, create two spatial weights files for the POLICE data set, one based on rook contiguity, the other on 5 nearest neighbors. Use `FIPNO` as the `ID` variable. Call these files `policerook.gal` and `policek5.gwt`, respectively. See the instructions in the*GeoDa* Workbook for details. Make sure you save (or copy) the files to the working directory for your R project.

2.3 Creating a Data Frame

2.3.1 Command Line

In R, the easiest way to create a data frame from a comma-delimited ascii file is to use a variant of the `read.table` function that takes into account the presence of the comma. This is the `read.csv` function. Check out `>help(read.csv)` to see all the details and options. The files created by *GeoDa* contain a header line (`header=TRUE`) and an extra first line that R doesn't need (the line with the `82,12`). You avoid reading this line by setting the `skip` option to 1. The proper command to read the file is then:

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

As usual, nothing seems to happen when you enter this, but you can check the presence of the new `police` object with a `summary` command:

```
> summary(police)
     FIPSNO          POLICE              POP                TAX
 Min.   :28001   Min.   :   49.0   Min.   :  2500   Min.   : 81.0
 1st Qu.:28042   1st Qu.:  291.5   1st Qu.: 13275   1st Qu.:106.2
 Median :28082   Median :  432.5   Median : 20700   Median :125.5
 Mean   :28082   Mean   :  927.8   Mean   : 31311   Mean   :142.2
 3rd Qu.:28122   3rd Qu.:  955.8   3rd Qu.: 35850   3rd Qu.:156.5
 Max.   :28163   Max.   :10971.0   Max.   :253100   Max.   :674.0
 ...
```

Also, an `>ls()` command will reveal `"police"` as one of the objects in your work space.

You can specify a variable to serve as the row name in the data frame, but I usually prefer not to do that. The option is `row.names="varname"`.

2.3.2 Writing a Function

The real power of R is obtained through writing functions that carry out repetitive actions. To practice writing functions, you will write a simple procedure to turn a *GeoDa* text output file into an R data frame. You will then "compile" it to make it part of your R environment. At that point, you can invoke it in the same way as any other built-in R function.

Use any text editor (e.g., Notepad or similar) to create a text file, say `read.geoda.R`. Even if you use a "real" word processor, make sure to save the file as *pure* text, so that no extra characters sneak in.

Enter the following lines (lines starting with # are comments and you can skip those to minimize typing):

```
# read.geoda
# helper function to read GeoDa export files
# specify input file = file
# default is no row names, specify row names as second parameter if needed
# example: balt <- read.geoda("baltim.txt")
#                  balt <- read.geoda("baltim.txt","STATION")

read.geoda <- function(file,row.names=NULL)
{
          read.csv(file=file,header=TRUE,skip=1,row.names=row.names)
}
```

The `read.geoda` function simply calls the standard `read.csv` function with some preset options. Note how the `file` is a required parameter, but the `row.names` can be omitted, since it is set to `NULL` as the default. The way the `row.names` is used in the argument list requires only that the name of the variable be specified, no need to write `row.names=`.

To make the function part of your environment, you must first "compile" it with the R `source` command. Make sure the file `read.geoda.R` is in your working directory and enter:

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

The `read.geoda` function is now available and can be invoked directly. Note that the `R` file extension is only for the text file that contains the code, the function as defined in that file does *not* have an `R` extension. So, don't use `read.geoda.R` as the function name. To practice, first remove the current `police` object using the `>rm(police)` command. Check that it is really gone by trying a `summary(police)`. You should get

```
> summary(police)
Error in summary(police) : Object "police" not found
```

Next, recreate the `police` object with:

```
> police <- read.geoda("police.txt")
```

You can again make sure it is there with a `summary` command or by using `ls()`.

2.4 Creating a Neighbor List from a GAL File

To manipulate spatial weights, you will need to load the spdep package. Of course, this assumes that you previously installed it (see the R tutorials on how to do this). To load the package, use the `library` command as illustrated below:

```
> library(spdep)
Loading required package: tripack
Loading required package: maptools
Loading required package: foreign
Loading required package: SparseM
[1] "SparseM library loaded"

Attaching package 'spdep':

    The following object(s) are masked _by_ .GlobalEnv :
```

```
    read.geoda

>
```

Note how the `read.geoda` function you just created conflicts with the function of the same name that is included in the latest version of `spdep` .[1] If you did not source `read.geoda.R`, you will not have the warning. Note that the functions to read spatial weights have undergone several changes since the earlier *Data* tutorial was written. Specifically, the revised `read.gal` function can now read new-style GAL weights files, that is, files created by *GeoDa*. This workbook uses the most current version of these functions.

2.4.1 Using read.gal

Check out the features and parameters of the new `read.gal` function with the `help(read.gal)` command. You will see that there are three parameters: the name of the input file, a `region.id` and an `override.id`. The main purpose of these parameters is to avoid having to assume that the data set (the data frame) and the weights file have their observations in the same order. In *GeoDa*, the `ID` variable creates a key that can be matched with a corresponding variable in a data set to ensure the correct order. This is not an issue when both the data set and the weights are created together by *GeoDa*, as you just did, but it can be a problem when this is not the case.

For our purposes, it will suffice to set the `override.id = TRUE`, rather than the default `FALSE` to force `spdep` to use the new *GeoDa* GAL weights format. For example, with the `policerook.GAL` file as input, this becomes:[2]

```
> polgal <- read.gal("policerook.GAL",override.id=TRUE)
```

This creates `polgal` as a neighbor list object. Technically, this is a list of lists, one for each observation, containing a list of the neighbors. In fact, it consists of more than just the neighbor information, and contains a list of `ID` values, and some metadata, as an `attributes` command reveals:

```
> attributes(polgal)
$class
```

[1]These functions are basically the same, except that the "offcial" version has `skip = 0` as the default. So, to use the `read.geoda` function from `spdep`, make sure to set the option `skip = 1`.

[2]Make sure to check the spelling of the file name, specifically, whether the GAL file extension is in caps or lowercase. `R` is case sensitive.

```
[1] "nb"

$region.id
 [1] "28003" "28141" "28139" "28009" "28033" "28093" "28143" "28137" "28117"
[10] "28145" "28071" "28107" "28027" "28081" "28119" "28057" "28115" "28161"
[19] "28135" "28013" "28011" "28095" "28017" "28133" "28043" "28083" "28025"
[28] "28087" "28155" "28015" "28097" "28105" "28151" "28019" "28051" "28053"
[37] "28007" "28103" "28159" "28125" "28163" "28055" "28099" "28079" "28069"
[46] "28089" "28123" "28149" "28121" "28075" "28101" "28049" "28061" "28023"
[55] "28129" "28021" "28127" "28029" "28153" "28063" "28067" "28031" "28065"
[64] "28077" "28001" "28085" "28037" "28091" "28073" "28035" "28111" "28041"
[73] "28157" "28005" "28113" "28147" "28109" "28039" "28131" "28059" "28047"
[82] "28045"

$GeoDa
$GeoDa$shpfile
[1] "police"

$GeoDa$ind
[1] "FIPSNO"


$gal
[1] TRUE

$call
[1] TRUE

$sym
[1] TRUE
```

When you `print` the neighbor list, you no longer get a complete listing of its contents, as was the case in earlier versions. Instead, a summary-like overview of its main characteristics is listed:

```
> print(polgal)
Neighbour list object:
Number of regions: 82
Number of nonzero links: 402
Percentage nonzero weights: 5.978584
```

```
Average number of links: 4.902439
```

This may change again in the future.

2.4.2 Weights Characteristics

The characteristics of the weights file are obtained with the usual `summary` command:

```
> summary(polgal)
Neighbour list object:
Number of regions: 82
Number of nonzero links: 402
Percentage nonzero weights: 5.978584
Average number of links: 4.902439
Link number distribution:

 2  3  4  5  6  7
 1 13 16 23 21  8
1 least connected region:
28045 with 2 links
8 most connected regions:
28145 28071 28135 28043 28155 28007 28163 28085 with 7 links
```

Note how the link number distribution and summary of the connectedness structure use the `region.id` values to identify the observations. If you use `Tools > Weights > Properties` in *GeoDa*, you will observe the same frequency distribution in the link histogram.

2.5 Creating a Neighbor List from a GWT File

The new function `read.gwt2nb` included in the latest versions of `spdep` (included since the *Data* tutorial) is designed to convert a GWT weights file into a neighbor list object.

2.5.1 Using read.gwt2nb

The function `read.gwt2nb` takes as arguments the file name for the GWT file and the name of an `ID` variable, `region.id`. The latter must be "available" in your workspace. This is achieved by "attaching" the data frame that contains the variable. In our case, this is the `police` data frame. By attaching the data frame, the variables can be accessed by their name, e.g., as FIPSNO, instead of the more cumbersome police$FIPSNO. Attach the police data frame as:

```
> attach(police)
```

Make sure it worked by trying a summary of a variable, for example:

```
> summary(POLICE)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   49.0   291.5   432.5   927.8   955.8 10970.0
```

Now, turn the policek5.GWT file into a neighbor list, using FIPSNO as the region.id, and followed by a summary, as:

```
> polgwt <- read.gwt2nb("policek5.GWT",region.id=FIPSNO)
> summary(polgwt)
Neighbour list object:
Number of regions: 82
Number of nonzero links: 410
Percentage nonzero weights: 6.097561
Average number of links: 5
Non-symmetric neighbours list
Link number distribution:

 5
82
82 least connected regions:
28003 28141 28139 28009 28033 28093 28143 28137
... 28039 28131 28059 28047 28045 with 5 links
82 most connected regions:
28003 28141 28139 28009 28033 28093 28143 28137
... 28039 28131 28059 28047 28045 with 5 links
```

2.5.2 Checking for Symmetry of a Neighbor List

A useful function is a check for the symmetry of the neighbor list. Spatial weights based on contiguity should be symmetric, whereas weights derived from a k-nearest neighbor relation are typically not symmetric.

The is.symmetric.nb function in spdep implements this check. It takes the name of the neighbor object as argument. In our example, for the GAL-based weights:

```
> print(is.symmetric.nb(polgal))
[1] TRUE
```

and for the GWT-based weights,

```
> print(is.symmetric.nb(polgwt))
Non matching contiguities: 2 and 14
Non matching contiguities: 3 and 4
Non matching contiguities: 6 and 17
...
Non matching contiguities: 82 and 79
Non-symmetric neighbours list
[1] FALSE
```

2.6 Practice

Try the creation of a data frame, GAL-based neighbor lists and GWT-based neighbor lists for one of the other polygon sample data sets, such as the St Louis homicide data (STL HOM). Alternatively, try it on your own data set.

Exercise 3

Spatial Autocorrelation Analysis in R

3.1 Objectives

The purpose of this exercise is to illustrate how descriptive spatial autocorrelation analysis can be carried out in the R package spdep. We will also begin exploring some basic programming techniques in R, and start using plotting commands to design customized graphs.

The examples below all use the built-in COLUMBUS data set from spdep. Before trying the commands, make sure spdep is loaded by typing:

```
> spdep()
[1] "spdep, version 0.3-8, 2004-12-29"
```

If spdep is not loaded, you will get an error message. Also, load the COLUMBUS data set and `attach` it so that you can refer to the variables by name:

```
> data(columbus)
> attach(columbus)
```

For the practice sessions, you will be using the data frame for the POLICE data set and the GAL based neighbor list constructed in Exercise 2.

3.2 Converting Neighbor Lists to Spatial Weights

The neighbor list (an object of class `nb`) considered in Exercise 2 is only one of several classes that handle spatial contiguity information in spdep. It is primarily intended to store information on the `ID` of the neighbor for each observation. For the spatial autocorrelation tests and spatial diagnostics in regression analysis, a different type of object is used, referred to as a *spatial weights object* (an object of class `listw`).

One converts an `nb` object into a `listw` object by means of the `nb2listw` function. This function has several options, and the defaults are not necessarily what one wants. An excerpt of `help(nb2listw)` shows the parameter list and the default settings:

```
Usage:

     nb2listw(neighbours, glist=NULL, style="W", zero.policy=FALSE)

Arguments:

neighbours: an object of class 'nb'
glist: list of general weights corresponding to neighbours
style: 'style' can take values W, B, C, U, and S
zero.policy: If FALSE stop with error for any empty neighbour sets, if
          TRUE permit the weights list to be formed with zero-length
          weights vectors
```

Two things to watch out for is to make sure that the "islands" are handled appropriately, since this affects future analysis. The default is to have any islands induce missing value codes through `zero.policy=FALSE`, which will stop the program when islands are encountered. To allow islands, make sure to explicitly set `zero.policy=TRUE`. As before, if you are happy with the default settings, there is no need to specify any of the parameters, and a simple `nb2listw(neighbor object)` will do.

3.2.1 Example

In the built-in COLUMBUS data set, the object `col.gal.nb` is a neighbor list for queen contiguity.[1] Since there are no islands in Columbus and the default of row-standardization (`style="W"`) is fine, we only need to specify the neighbor list name:

```
> colqueen <- nb2listw(col.gal.nb)
```

[1]Note that the queen contiguity weights are the ones originally included with the COLUMBUS data set on the now deprecated `geog55.geog.uiuc.edu` ftp site. These weights differ slightly from the ones constructed by *GeoDa*.

A check on the `class` of `colqueen` reveals that it is indeed an object of `class listw`:

```
> class(colqueen)
[1] "listw" "nb"
```

and `summary` gives the same connectedness information as before for the `nb` object. However, note how several summary measures of the weights have been precomputed, such as the various matrix traces used in the test statistics, more precisely, `nn, S0, S1` and `S2`:

```
> summary(colqueen)
Characteristics of weights list object:
Neighbour list object:
Number of regions: 49
Number of nonzero links: 230
Percentage nonzero weights: 9.579342
Average number of links: 4.693878
Link number distribution:

 2  3  4  5  6  7  8  9 10
 7  7 13  4  9  6  1  1  1
7 least connected regions:
1005 1008 1045 1047 1049 1048 1015 with 2 links
1 most connected region:
1017 with 10 links

Weights style: W
Weights constants summary:
   n   nn S0       S1       S2
W 49 2401 49 23.48489 204.6687
```

Finally, to see the actual weights contained in the object, use `colqueen$weights`, which is a list of lists of spatial weights for each observation:

```
> colqueen$weights
[[1]]
[1] 0.5 0.5

[[2]]
[1] 0.3333333 0.3333333 0.3333333
```

```
[[3]]
[1] 0.25 0.25 0.25 0.25
...
```

3.2.2 Practice

Use the `polgal` neighbor list object created in Exercise 2 for the POLICE data set to turn it into a `listw` object and check its characteristics. If you did not save `polgal` in your work space, you will need to recreate it from the `policerook.GAL` file (follow the directions in in Section 2.4.1 on p. 8). The results are listed below.

Creating a spatial weights list

```
> polrook <- nb2listw(polgal)
> summary(polrook)
Characteristics of weights list object:
Neighbour list object:
Number of regions: 82
Number of nonzero links: 402
Percentage nonzero weights: 5.978584
Average number of links: 4.902439
Link number distribution:

 2  3  4  5  6  7
 1 13 16 23 21  8
1 least connected region:
28045 with 2 links
8 most connected regions:
28145 28071 28135 28043 28155 28007 28163 28085 with 7 links

Weights style: W
Weights constants summary:
   n   nn S0       S1       S2
W 82 6724 82 35.14864 332.3231
> polrook$weights
[[1]]
[1] 0.3333333 0.3333333 0.3333333

[[2]]
```

```
[1] 0.3333333 0.3333333 0.3333333

[[3]]
[1] 0.25 0.25 0.25 0.25
...
```

3.3 Moran's I

Moran's I test statistic for spatial autocorrelation is implemented in `spdep`. There are two separate functions, `moran.test`, where inference is based on a normal or randomization assumption, and `moran.mc`, for a permutation-based test. Both take a variable name or numeric vector and a spatial weights list object (`listw`), in that order, as mandatory parameters. The permutation test also requires the number of permutations as a third mandatory parameter. Both functions also have several options, discussed in more detail below.

3.3.1 Normal and Randomization Inference

The different parameters and options for `moran.test` are revealed by a call to `help(moran.test)`:

```
     moran.test(x, listw, randomisation=TRUE, zero.policy=FALSE,
             alternative="greater", rank = FALSE,
             na.action=na.fail, spChk=NULL)

Arguments:

x: a numeric vector the same length as the neighbours list in
    listw
listw: a 'listw' object created for example by 'nb2listw'
randomisation: variance of I calculated under the assumption of
          randomisation, if FALSE normality
zero.policy: if TRUE assign zero to the lagged value of zones
          without neighbours, if FALSE assign NA
alternative: a character string specifying the alternative
          hypothesis, must be one of greater (default), less
          or two.sided.
rank: logical value - default FALSE for continuous variables,
          if TRUE, uses the adaptation of Moran's I for ranks
          suggested by Cliff and Ord (1981, p. 46)
na.action: a function (default na.fail), can also be na.omit or
               na.exclude - in these cases the weights list will
               be subsetted to remove NAs in the data. It may be
               necessary to set zero.policy to TRUE
               because this subsetting may create no-neighbour
               observations.
               Note that only weights lists created without using
               the glist argument to nb2listw may be subsetted.
               If na.pass is used, zero is substituted for NA
               values in calculating the spatial lag
spChk: should the data vector names be checked against the
          spatial objects for identity integrity, TRUE, or FALSE,
          default NULL to use 'get.spChkOption()'
```

Of the optional parameters, two are very important. The `randomisation` option (watch the spelling!) is set to `TRUE` by default, which implies that in order to get inference based on a normal approximation, it must be explicitly set to `FALSE`. Similarly, the default is a *one-sided* test, so that in order to obtain the results for the (more commonly used) two-sided test, the option `alternative` must be set explicitly to `"two.sided"`. Note also that the `zero.policy` option is set to `FALSE` by default, which means that islands result in a missing value code (`NA`). Setting this option to `TRUE` will set the spatial lag for islands to the customary zero value.

To illustrate this, use the variable `CRIME` and the weights list `colqueen` with the normal approximation in a two-sided test:[2]

```
> moran.test(CRIME,colqueen,randomisation=FALSE,
+  alternative="two.sided")

        Moran's I test under normality

data:  CRIME
weights: colqueen

Moran I statistic standard deviate = 5.3818, p-value = 7.374e-08
alternative hypothesis: two.sided
sample estimates:
```

[2]Make sure you have the `columbus` data set attached, so that the variables can be referred to by name.

```
Moran I statistic       Expectation          Variance
      0.485770914      -0.020833333       0.008860962
```

Note how, unlike previous practice, the “object” created by `moran.test` was not assigned to a specific variable. If you simply want to get the results, this is not necessary. By entering the test this way, you indirectly invoke the `print` function for the object. If you want to access the individual results of the test for further processing, you should assign the `moran.test` to an object and then print that object. For example, using `INC`:

```
> mornorINC <- moran.test(INC,colqueen,randomisation=FALSE,
+ alternative="two.sided")
> print(mornorINC)

        Moran's I test under normality

data:  INC
weights: colqueen

Moran I statistic standard deviate = 4.6495, p-value = 3.327e-06
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.416837942      -0.020833333       0.008860962
```

Note how the expectation and variance of the Moran's I test statistic is the same for both `CRIME` and `INC`. In fact, under the normal approximation, these moments only depend on the spatial weights, and not on the variable under consideration.

The `mornorINC` object belongs to the class `htest`, which is a generic class in R designed to hold the results of test statistics. Technically, it is a list and all its items can be accessed individually (provided you know what they are called; check the help). For example,

```
> class(mornorINC)
[1] "htest"
> mornorINC$statistic
Moran I statistic standard deviate
                          4.649514
```

shows the information stored in the `statistic` element of the list.

When the null hypothesis considered is based on the randomization distribution, the `randomisation` option does not need to be set. For example, again using `CRIME`:

```
> moran.test(CRIME,colqueen,alternative="two.sided")

        Moran's I test under randomisation

data:  CRIME
weights: colqueen

Moran I statistic standard deviate = 5.3427, p-value = 9.157e-08
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.485770914      -0.020833333       0.008991121
```

Note how the value of the statistic and its expectation do not change relative to the normal case, only the variance is different (and thus the z-value and associated p-value).

Finally, leaving all the defaults as they are:

```
> moran.test(CRIME,colqueen)

        Moran's I test under randomisation

data:  CRIME
weights: colqueen

Moran I statistic standard deviate = 5.3427, p-value = 4.578e-08
alternative hypothesis: greater
sample estimates:
Moran I statistic       Expectation          Variance
      0.485770914      -0.020833333       0.008991121
```

where the only thing different is the p-value (half the p-value of the two-sided test).

3.3.2 Permutation Inference

A Moran's I test statistic with inference based on random permutation is contained in the function `moran.mc`. As in `moran.test`, it takes the variable name (or vector) and the weights list file (`listw` object) as the first two mandatory arguments. It also needs the number of permutations as the third argument (`nsim`). Since the rank of the observed statistic is computed relative to the reference distribution of statistics for the permuted data sets, it is good practice to set this number to something ending on 9 (such as 99 or 999). This will yield nicely rounded pseudo p-values (like 0.01 or 0.001). The other options are identical to those for the `moran.test` function.

For example, using the `CRIME` variable with the `colqueen` weights, setting the number of permutations to 99 and leaving everything else to the default (e.g., a one-sided test):

```
> morpermCRIME <- moran.mc(CRIME,colqueen,99)
> morpermCRIME

        Monte-Carlo simulation of Moran's I

data:  CRIME
weights: colqueen
number of simulations + 1: 100

statistic = 0.4858, observed rank = 100, p-value = 0.01
alternative hypothesis: greater
```

Note how the number of simulations + 1 is a round 100. None of the permuted data sets yielded a Moran's I greater than the observed value of 0.4858, hence a pseudo p-value of 0.01.[3]

In this example, the results of the procedure were assigned to an object of class `htest` (`morpermCRIME`), which must be printed to reveal its contents. One item in the list `morpermCRIME` is a vector with the computed statistics for each of the permuted data set. This is contained in the item `$res`, as in:

```
> morpermCRIME$res
[1] -0.1211582450  0.0250296894 ... -0.0192339594 -0.1944962763
[6]  0.0397742260 -0.0554032219 ...  0.1625140071 -0.1785841236
  ...
[91]  0.0196008271 -0.1764712321 ... -0.0832054684  0.0782020854
[96]  0.0910332773 -0.1760530540 ... -0.0806383188  0.4857709137
```

Note that the "observed" value of Moran's I (0.48577) is included in this list as the last element. By default, the contents of this vector will slightly differ

[3]This is obtained as the ratio of the number of values greater or equal to observed statistic + 1 over the number of simulations + 1. In the example, this is (0+1)/(99+1)=0.01.

from run to run, since they are based on random permutations. The default random number seed value is determined from the current time and so no random permuation will be identical. To control the seed, use the R function `set.seed(seed, kind = NULL)` right before invoking the `moran.mc` command, and set the same value each time. For example, try this using `set.seed(123456)`:

```
> set.seed(123456)
> morpermCRIME <- moran.mc(CRIME,colqueen,99)
> morpermCRIME$res
  [1] -6.607790e-02  1.398150e-02  7.363572e-02  ...
  ...
  [100]  4.857709e-01
```

Unless you want to replicate results exactly, it is typically better to let the randomness of the clock set the seed.

3.3.3 Plotting the Reference Distribution

The full vector of Moran statistics for the permuted data sets lends itself well to a histogram or density plot. R has very sophisticated plotting functions, and this example will only scratch the surface. Consult the help files for further details and specific options.

In order to construct a density plot, we first need to create a "density" object. To make sure that this is the reference distribution plotted for the randomly permuted data sets only, we must remove the last element from `morpermCRIME$res`:

```
> morp <- morpermCRIME$res[1:length(morpermCRIME$res)-1]
```

Next, we must pass the new list (`morp`) to the function `density`. This function has many options, and in this example they are all kept to their default settings (check out `help(density)` for further details), so that only the vector with statistics is specified. The result is stored in the object `zz`:

```
> zz <- density(morp)
```

Next, we will plot three graphs on top of each other: a (continuous) density function (based on `zz`), a histogram for the reference distribution, and a line indicating the observed Moran's I. The latter is contained in the `statistic` attribute of the `moran.mc` object, `morpermCRIME$statistic`.

In the code that follows, the density plot is drawn first, along with the titles (`main` on top and `xlab` under the x-axis).[4] Also, to make the plot more distinct, a double line width is set (`lwd=2`) and the plot is in red (`col=2`). Finally, to make sure that the default settings for the plot do not make it too small, we explicitly set the horizontal (`xlim`) and vertical (`ylim`) width (typically, this takes some trial and error to get it right):

```
> plot.density(zz,main="Moran's I Permutation Test",
+ xlab="Reference Distribution",xlim=c(-0.3,0.7),
+ ylim=c(0,6),lwd=2,col=2)
```

Next, you add the histogram (`hist`) and a vertical line (`abline`). The histogram is drawn in regular thickness black lines, while the vertical line is double thickness (`lwd=2`) and drawn in blue (`col=4`):

```
> hist(morp,freq=F,add=T)
> abline(v=morpermCRIME$statistic,lwd=2,col=4)
```

The result is as in Figure 3.1 on p. 24.

Note that the example as described will plot the figure on your screen. In order to create hard copy output, you need to specify an output `device`. R supports several devices, but by far the most commonly used are the `postscript` and `pdf` devices. You must first open the device explicitly. For example,

```
>postscript(file="filename")
```

for a postscript file, or

```
>pdf(file="filename")
```

for a pdf file. All plotting commands that follow will be written to that device, until you close it with a `dev.off()` command. For example,

```
> postscript(file="moran.pdf")     # file name for pdf file
...                                # plotting commands
> dev.off()                        # close postscript device
```

[4]Several, but not all the plotting functions in R support an `add=T` argument, which adds a graph to an existing plot. Since the current version of `plot.density` does not support this argument, while the other two plots do, it has to be drawn first.

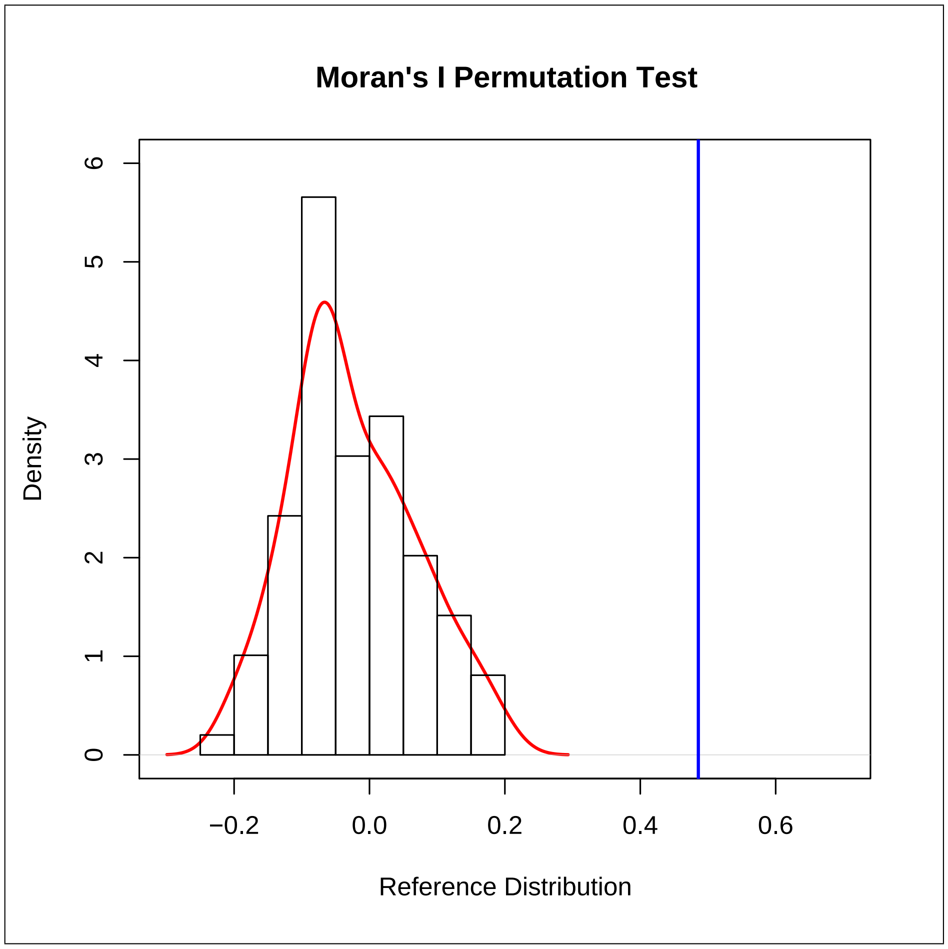

Figure 3.1: Moran's I Permutation Test Plot.

3.3.4 Practice

Use the POLICE data set and the `polrook` weights to test for spatial autocorrelation in the `CRIME` variable for the Mississippi counties. Use both the normal and randomization assumptions, as well as a permutation approach. Try making a graph of the distribution of the statistic in the simulated data sets, as in Figure 3.1.

Make sure to `detach(columbus)` and `attach(police)` before carrying out the tests so that you use the correct variable. Afterwards, do the reverse to get back to the COLUMBUS data for the remainder of the illustrations.

The results for the statistics are given below.

Moran's I, Normal Assumption

```
> moran.test(CRIME,polrook,randomisation=FALSE,
+ alternative="two.sided")

        Moran's I test under normality

data:  CRIME
weights: polrook

Moran I statistic standard deviate = 1.9108, p-value = 0.05604
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.121668149      -0.012345679       0.004919118
```

Moran's I, Randomization Assumption

```
> moran.test(CRIME,polrook,alternative="two.sided")

        Moran's I test under randomisation

data:  CRIME
weights: polrook

Moran I statistic standard deviate = 2.4522, p-value = 0.01420
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.121668149      -0.012345679       0.002986579
```

Moran's I, Permutation Approach

```
> morMissCRIME <- moran.mc(CRIME,polrook,999)
> print(morMissCRIME)

        Monte-Carlo simulation of Moran's I

data:  CRIME
```

```
weights: polrook
number of simulations + 1: 1000

statistic = 0.1217, observed rank = 986, p-value = 0.014
alternative hypothesis: greater
```

3.4 Constructing a Spatially Lagged Variable

Spatially lagged variables are important elements of many spatial regression specifications. In `spdep`, they are constructed by means of the `lag.listw` function. This function takes as arguments a spatial weights object of class `listw` and a conforming matrix (i.e., a matrix with the same number of rows as the dimension of the spatial weights).[5] Note that the documentation for `lag.listw` only refers to a "numeric vector," but this is incomplete. A look at the source code (type `lag.listw` without the usual parentheses) reveals that it works equally well for a matrix, i.e., for several variables in one function call. The third (optional) argument to `lag.listw` is the `zero.policy`, which we can ignore for the COLUMBUS and POLICE sample data sets.

3.4.1 Example

First, we will combine the three variables `CRIME`, `INC` and `HOVAL` into a matrix.[6] This is accomplished by means of the `cbind` command and will be an often used procedure when you program your own routines based on matrix algebra. For example, you create the matrix `xx` as:

```
> xx <- cbind(CRIME,INC,HOVAL)
```

and check the contents with

```
> xx
         CRIME    INC  HOVAL
 [1,] 15.725980 19.531 80.467
 [2,] 18.801754 21.232 44.567
 [3,] 30.626781 15.956 26.350
 ...
```

[5]The order of these arguments is the *reverse* of that in the Moran test statistics. Here, the weights object must come first, and the vector or matrix second. This order *must* be respected.

[6]Make sure you detached the `police` data frame and attached `columbus`.

Now, you can construct a spatially lagged version of this matrix, using the `colqueen` weights:

```
> wxx <- lag.listw(colqueen,xx)
```

and check the contents

```
> wxx
          [,1]      [,2]     [,3]
 [1,] 24.71427 18.594000 35.45850
 [2,] 26.24684 13.321333 46.67233
 [3,] 29.41175 14.123000 45.36475
 ...
```

Since this is a fairly small data set, we can easily check if the spatial lag terms are what they should be. Extract the neighbor IDs and spatial weights from `listw` by referring to the proper list elements (a `listw` object is a list, i.e., a combination of different elements). You see the names of the elements as the result of an `attributes` command (this lists the "attributes" of an object). For example,

```
> attributes(colqueen)
$names
[1] "style"      "neighbours" "weights"

$class
[1] "listw" "nb"

$region.id
 [1] 1005 1001 1006 1002 ... 1037 1039 1040 1009
...
[46] 1048 1015 1027 1026

$call
nb2listw(neighbours = col.gal.nb)
```

This shows that the neighbor IDs are in `colqueen$neighbours` (spelling!) and the corresponding weights in `colqueen$weights`. Specifically, for the first observation:

```
> colqueen$neighbours[1]
[[1]]
[1] 2 3
```

and

```
> colqueen$weights[1]
[[1]]
[1] 0.5 0.5
```

A simple calculation yields the spatial lags for the first observation "by hand":

```
> (xx[2,] + xx[3,])/2.0
   CRIME      INC    HOVAL
24.71427 18.59400 35.45850
```

which are the same values as in the first row of `wxx`.

Compare the descriptive characteristics of the original variables to their spatial lags:

```
> summary(xx)
     CRIME               INC              HOVAL
 Min.   : 0.1783   Min.   : 4.477   Min.   :17.90
 1st Qu.:20.0485   1st Qu.: 9.963   1st Qu.:25.70
 Median :34.0008   Median :13.380   Median :33.50
 Mean   :35.1288   Mean   :14.375   Mean   :38.44
 3rd Qu.:48.5855   3rd Qu.:18.324   3rd Qu.:43.30
 Max.   :68.8920   Max.   :31.070   Max.   :96.40
> summary(wxx)
       X1              X2                X3
 Min.   :13.85   Min.   : 8.539   Min.   :23.41
 1st Qu.:24.71   1st Qu.:10.957   1st Qu.:30.82
 Median :35.90   Median :14.303   Median :36.38
 Mean   :34.88   Mean   :14.742   Mean   :38.73
 3rd Qu.:45.39   3rd Qu.:17.161   3rd Qu.:45.36
 Max.   :54.91   Max.   :26.066   Max.   :75.13
```

Note how the spatial lag operation compresses the distribution of the variable. The measures of "spread" are substantially smaller than for the originals.

3.4.2 Practice

Compute the spatial lags for the `CRIME`, `POLICE` and `INC` variables in the `police` data set, using the `polrook` spatial weights. Compare the descriptive statistics of the original variables to their spatial lags. Experiment with other spatial weights and/or try some statistical plots to visualize the difference between the distributions.

The basic results are given below.

Spatial Lags

```
> policex <- cbind(CRIME,POLICE,INC)
> wpolicex <- lag.listw(polrook,policex)
> summary(policex)
     CRIME            POLICE              INC       
 Min.   :   5.0   Min.   :   49.0   Min.   : 4836  
 1st Qu.:  76.5   1st Qu.:  291.5   1st Qu.: 6520  
 Median : 141.5   Median :  432.5   Median : 7008  
 Mean   : 194.5   Mean   :  927.8   Mean   : 7198  
 3rd Qu.: 298.0   3rd Qu.:  955.8   3rd Qu.: 7612  
 Max.   :1739.0   Max.   :10971.0   Max.   :10506  
> summary(wpolicex)
       X1              X2                X3      
 Min.   : 38.5   Min.   : 225.0   Min.   :6109  
 1st Qu.:125.4   1st Qu.: 469.0   1st Qu.:6847  
 Median :163.0   Median : 690.5   Median :7092  
 Mean   :191.8   Mean   : 931.5   Mean   :7191  
 3rd Qu.:224.6   3rd Qu.: 991.1   3rd Qu.:7449  
 Max.   :732.7   Max.   :3434.0   Max.   :8608  
```

3.5 Moran Scatter Plot

The Moran scatter plot is also included in the `spdep` functionality. It is implemented in the function `moran.plot`. This takes as arguments a variable name, *followed by* a spatial weights objects of class `listw`. Optional parameters are the `zero.policy` as well as several ways to fine tune the labeling of high influence points, the x and y-axes, and the various graphical parameters (passed as in `par(..)`). The function `moran.plot` also creates a so-called "influence" object from the `influence.measures` passed back from the regression model that is used to estimate the slope (Moran's I). Setting the option `quiet=TRUE` suppresses the influence measures (the default is `quiet=FALSE`).

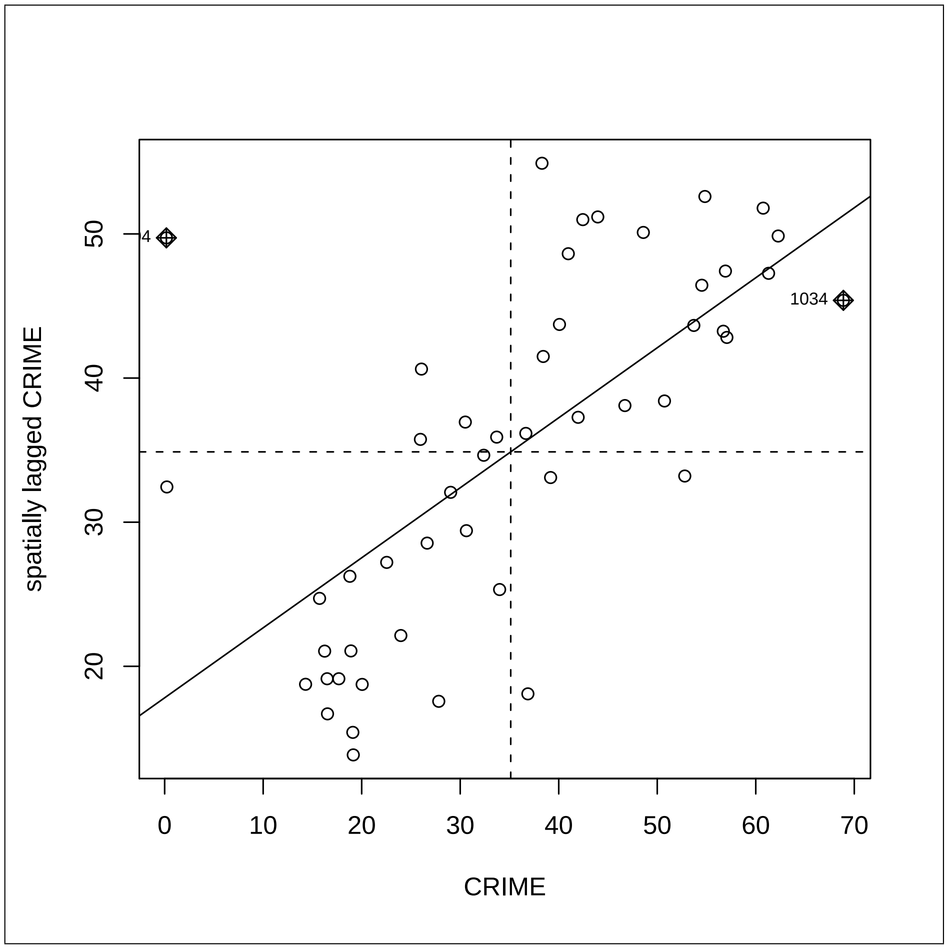

Figure 3.2: Moran Scatter Plot for `CRIME`, using `colqueen`.

3.5.1 Example

Use the `CRIME` variable from the `columbus` data frame and the spatial weights `colqueen` to invoke the Moran scatterplot, leaving all other settings to their default:

```
> moran.plot(CRIME,colqueen)
Potentially influential observations of
         lm(formula = wx ~ x) :

     dfb.1_  dfb.x   dffit   cov.r   cook.d  hat
```

```
1004  1.55_* -1.40_*  1.55_*  0.59_*  0.86_*  0.11
1034  0.15   -0.21   -0.23    1.14_*  0.03    0.11
```

Note the list of influence characteristics and the IDs of the influential observations. The plot is as in Figure 3.2 on p. 30.

The Moran scatterplot produced by spdep differs somewhat from the one implemented in *GeoDa*. Most importantly, spdep uses the original variables, whereas *GeoDa* constructs the plot for standardized variates. Also, it is not clear how Moran's I can be extracted from the plot without an additional computation. The object returned by the function is a matrix with the observations as rows and the six influence measures as columns.

3.5.2 Customizing the Moran Scatter Plot

We will program a simple function that constructs a Moran scatter plot using standardized variables and shows the value of Moran's I on the plot. We will use the spdep function `lag.listw` to build the spatial lags and the generic R regression function `lm` to get the coefficients. First we test the various commands by trying them step by step, with hard-coded variables, then we generalize and turn them into a function.

First set the generic variable `x` to the variable of interest (`CRIME`), and convert it to a standardized variate, with a simple check to make sure we did it right:

```
> x <- CRIME
> zx <- (x - mean(x))/sd(x)
> mean(zx)
[1] -1.948555e-16
> var(zx)
[1] 1
```

Next, create the spatial lag for the standardized variable, after turning the weights into a generic variable:

```
> wfile <- colqueen
> wzx <- lag.listw(wfile,zx)
```

Now, compute the intercept and slope (Moran's I), using `lm`:

```
> morlm <- lm(wzx ~ zx)
> aa <- morlm$coefficients[1]
> mori <- morlm$coefficients[2]
```

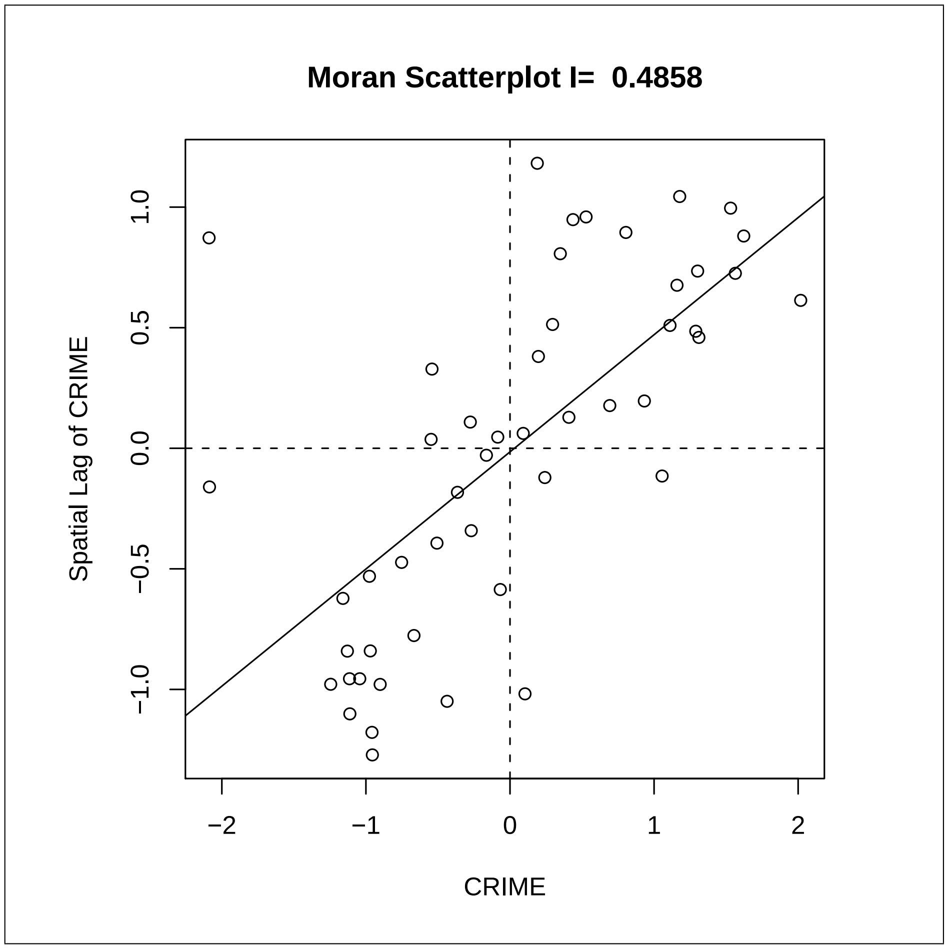

Figure 3.3: Moran Scatter Plot for standardized `CRIME`, using `colqueen`.

```
> aa
(Intercept)
-0.01496451
> mori
       zx
0.4857709
```

Finally, we set up the plot as a simple scatterplot (`plot(x,y)`), add the regression line and the two axes (using `abline`), label the axes (using `xlab` and `ylab`), and put a title containing the value of Moran's I at the top (using `title`, the `paste` command to put together text fragments, and a combination of `format` and `round` to make sure the Moran's I doesn't have too many decimal digits):

```
> par(pty="s")    # make sure it's square
> plot(zx,wzx,xlab="CRIME",ylab="Spatial Lag of CRIME")
> abline(aa,mori)
> abline(h=0,lty=2)
> abline(v=0,lty=2)
> title(paste("Moran Scatterplot I= ",format(round(mori,4))))
```

The result looks like Figure 3.3 on p. 32.

To combine these commands into a function, create a file in any text editor and name it `moran.plot2.R`. Enter the commands listed below and save the file as a text file in your working directory.

```
# moran.plot2
# function for Moran scatterplot using standardized variates
# specify variable = x
# specify spatial weights (listw object) = wfile
# all other options are hard coded (so far)
# example: moran.plot2(CRIME,colqueen)
#

moran.plot2 <- function(x,wfile)
{
    xname <- deparse(substitute(x))    # get name of variable
    zx <- (x - mean(x))/sd(x)
    wzx <- lag.listw(wfile,zx)
    morlm <- lm(wzx ~ zx)
    aa <- morlm$coefficients[1]
    mori <- morlm$coefficients[2]
    par(pty="s")
    plot(zx,wzx,xlab=xname,ylab=paste("Spatial Lag of ",xname))
    abline(aa,mori,col=2)
    abline(h=0,lty=2,col=4)
    abline(v=0,lty=2,col=4)
    title(paste("Moran Scatterplot I= ",format(round(mori,4))))
}
```

The only things different from before are the use of the functions `deparse` and `substitute` to extract the variable name for use in the plot labels `xlab` and `ylab`. Source the function as

```
> source("moran.plot2.R")
```

and invoke it as `moran.plot2`. For example, passing `CRIME` and `colqueen` as the parameters:

```
> moran.plot2(CRIME,colqueen)
```

will yield the same plot as in Figure 3.3 (except that the regression line is in red and the axes drawn in blue, see Figure 3.4 on p. 35).

3.5.3 Practice

Try out the new function using the `police` data frame (make sure to first `detach(columbus)` then `attach(police)`), and construct a Moran scatterplot for the variable `INC` using `polrook` as the weight file. Compare this plot to the standard `spdep` `moran.plot`. Try some other variables and/or spatial weights. Also try to get your plot written out to a postscript or pdf file (make sure to include `dev.off()` at the end to switch back to screen output).

If you feel adventurous, check out the source code of `moran.plot` and add the various safety checks and the diagnostics of high leverage to `moran.plot2`.

Moran Scatter Plot

```
> pdf(file="morscatf.pdf")
> moran.plot2(INC,polrook)
> dev.off()
```

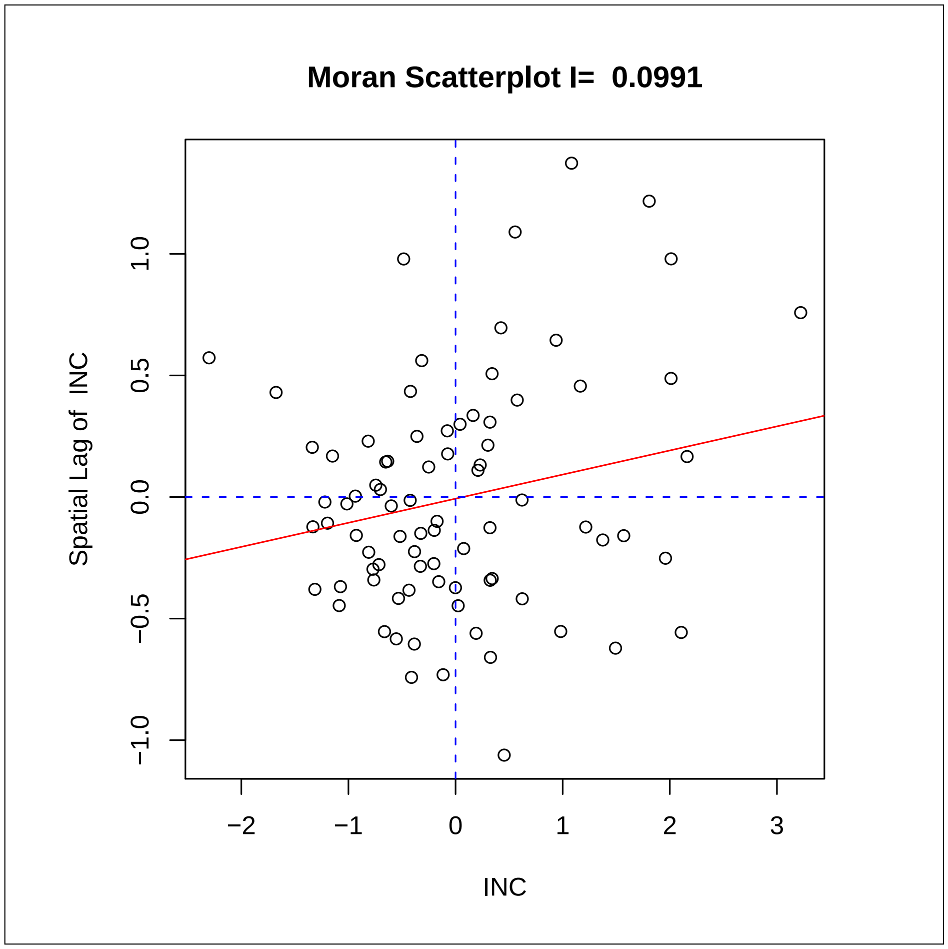

Figure 3.4: Moran Scatter Plot for standardized `INC`, using `polrook`.

Exercise 4

Monte Carlo Simulation (1) Assessing the Properties of a Test Statistic

4.1 Objectives

The purpose of this exercise is to illustrate how to set up a simple Monte Carlo simulation experiment to assess some properties of a test statistic. Specifically, the end goal is to describe the distribution and compare the nominal and empirical rejection frequencies for Moran's I under the null hypothesis. In the process, we will look at how to generate random variables and carry out basic matrix algebra in R. We will also use spdep to create spatial weights for regular grids and convert spatial weights to actual matrix objects.

As before, make sure spdep is loaded before starting the exercises. Since all the data will be generated, there is no need to specify a particular data frame.

4.2 Generating Random Variables

R has extensive functionality to generate random variables for a wide range of distributions and to compute probabilities for those distributions.[1]

Each distribution function has an R name, such as `norm` for the normal and `unif` for the uniform. The different random variable related functions

[1]See Chapter 8 on Probability Distributions in Venables et al. (2004) for further details.

are indicated by a single letter prefix to the distribution name. This includes `d` for the density, `p` for the cumulative density function, `q` for the quantile function, and `r` for simulation of random variables.[2] The arguments to these functions are, respectively, the value of a random variable `x` for `dxxx` and `pxxx`, a probability for `qxxx`, and the sample size for `rxxx`.

4.2.1 Example

To generate a vector of 25 standard normal variates, use `rnorm` with only the sample size as parameter, as in:

```
> x <- rnorm(25)
```

Check the mean and standard deviation:[3]

```
> mean(x)
[1] -0.199015
> sd(x)
[1] 0.9869352
```

Note how for a sample this small, the mean is fairly far from its theoretical expectation. To get a much closer fit and eliminate some of the extra variability induced by the random number generator, you need to generate much larger sets. For example:

```
> x <- rnorm(100000)
> mean(x)
[1] -0.005083976
> sd(x)
[1] 0.996822
> x <- rnorm(1000000)
> mean(x)
[1] -0.001698689
> sd(x)
[1] 1.000263
```

To generate a vector of normal random variates with a given mean and standard deviation, you need to pass these as arguments in addition to the

[2]The CDF and quantile functions have the special arguments `lower.tail` and `log` that are very useful in practice. For example, to compute the upper tail of a distribution as when the p-value of a statistic is required, or, 1 - CDF(x), set `lower.tail = FALSE`.

[3]Note that your results will vary, since the seed of the random number generator was not fixed using `set.seed`.

sample size. Note that you pass the *standard deviation*, **not** the variance. For example:

```
> x <- rnorm(1000,5,20)
> mean(x)
[1] 5.241835
> sd(x)
[1] 19.59694
```

You can now compare a given percentile in the generated distribution to its theoretical value. For example, in the 1000 generated values above the 95th percentile will be in row 950 of the sorted vector `x`:

```
> z <- sort(x)
> z[950]
[1] 36.46587
```

The theoretical value is found with the `qnorm` function, passing 0.95 (the probability) as the argument, in addition to the mean and standard deviation:

```
> qz <- qnorm(0.95,5,20)
> qz
[1] 37.89707
```

A plot of the sorted z values against qz values for the same percentile would yield a q-q plot. Conversely, finding the cumulative probability for `z[950]`:

```
> pz <- pnorm(z[950],5,20)
> pz
[1] 0.9421746
```

4.2.2 Practice

Experiment with some of the other distributions, such as the uniform, using `runif(n,min,max)`, or the Poisson, using `rpois(n,lambda)`. Use the techniques from Exercise 3 to draw a histogram or density plot for the generated variates and summarize the characteristics of the distribution (`mean`, `sd`, and `summary`).

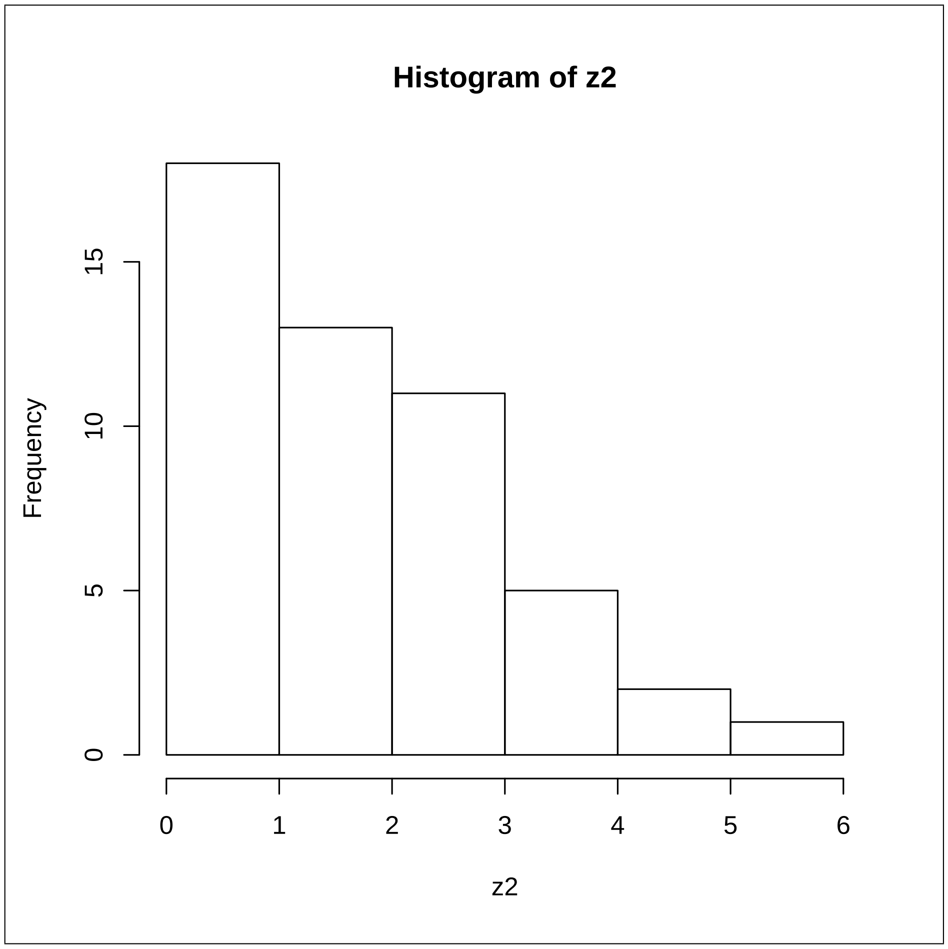

Figure 4.1: Histogram for Randomly Generated Poisson Variates.

Histogram for Poisson Variates

```
> z2 <- rpois(50,2)
> summary(z2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    1.00    2.00    2.14    3.00    6.00
> mean(z2)
[1] 2.14
> var(z2)
[1] 2.041224
```

```
> pdf(file="poisson.pdf")
> hist(z2)
> dev.off()
```

4.3 Basic Matrix Algebra Operations

Operations on matrices and vectors are element by element in R. So, if `x` is a vector (say `n` by 1) and `A` is a `n` by `n` matrix, then `A * x` will multiply each row element in `A` with the matching row element in `x` and yield another `n` by n matrix, not a n by 1 vector. To illustrate, use the `c` operator (for concatenate) to create a 5 by 1 vector `x` (by default this is a column vector), and the `matrix` function with row and column parameters to create a 5 by 5 matrix (of sequence numbers):

```
> x <- c(1:5)
> x
[1] 1 2 3 4 5
> A <- matrix(1:25,5,5)
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Note how the default ordering of a matrix is in so-called "column major order" (down each column first). Alternatively, a vector can be passed to the `array` function with a `dim` attribute for the same effect (this is useful for reshaping matrices as well):

```
> z <- c(1:25)
> B <- array(z,dim=c(5,5))
> B
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Now, multiply `A` by `x`:

```
> zz <- A * x
> zz
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    4   14   24   34   44
[3,]    9   24   39   54   69
[4,]   16   36   56   76   96
[5,]   25   50   75  100  125
```

In order to get *true* matrix multiplication, following to the rules of matrix algebra, a different operator must be used, symbolized as `%*%`:

```
> zz1 <- A %*% x
> zz1
     [,1]
[1,]  215
[2,]  230
[3,]  245
[4,]  260
[5,]  275
```

A few other important basic matrix operations are:

- `nrow(matrix)` for the number of rows
- `ncol(matrix)` for the number of columns
- `t(matrix)` to construct the transpose of a matrix, e.g., X'
- `diag(matrix)` to extract the diagonal elements from a matrix
- `diag(vector)` to create a diagonal matrix with `vector` as diagonal
- `crossprod(matrix1,matrix2)` to compute the crossproduct (e.g., $X'y$) between two matrices efficiently
- `%o%` for the outer product of two vectors, e.g., xx'
- `outer(vector1,vector2,function)` for any function applied to each pair x_i and y_j

4.3.1 Example

Start with a vector of 30 standard normal variates and convert it to a 10 by 3 matrix. Note how you can force "row major order" by setting the argument `byrow=TRUE`:

```
> x1 <- rnorm(30)
> x1
 [1]  1.86263657 -0.14248903 -1.26807899  ... -0.30481949
 [7]  1.05654040  0.46209724 -0.75313876  ... -1.95010825
[13] -0.33947593  0.55823272 -1.42072261  ...  0.37998187
[19] -0.75149464  0.47130272  0.76668664  ...  0.64171921
[25] -0.99857864 -1.53179763  1.94797343  ... -0.32623853
> XX <- matrix(x1,10,3)
> XX
            [,1]        [,2]       [,3]
 [1,]  1.8626366  1.02988875  0.7666866
 [2,] -0.1424890 -1.95010825  0.3450489
 [3,] -1.2680790 -0.33947593  0.6046427
 [4,]  0.2091366  0.55823272  0.6417192
 [5,]  0.4177341 -1.42072261 -0.9985786
 [6,] -0.3048195  0.06214715 -1.5317976
 [7,]  1.0565404 -0.31582345  1.9479734
 [8,]  0.4620972  0.37998187 -0.9578326
 [9,] -0.7531388 -0.75149464 -1.5308449
[10,]  0.7728531  0.47130272 -0.3262385
> XX <- matrix(x1,10,3,byrow=TRUE)
> XX
             [,1]       [,2]       [,3]
 [1,]  1.86263657 -0.1424890 -1.2680790
 [2,]  0.20913657  0.4177341 -0.3048195
 [3,]  1.05654040  0.4620972 -0.7531388
 [4,]  0.77285309  1.0298887 -1.9501082
 [5,] -0.33947593  0.5582327 -1.4207226
 [6,]  0.06214715 -0.3158235  0.3799819
 [7,] -0.75149464  0.4713027  0.7666866
 [8,]  0.34504894  0.6046427  0.6417192
 [9,] -0.99857864 -1.5317976  1.9479734
[10,] -0.95783257 -1.5308449 -0.3262385
```

Check the number of rows and columns and construct the transpose of matrix XX:

```
> rXX <- nrow(XX)
> rXX
[1] 10
> cXX <- ncol(XX)
> cXX
[1] 3
> tXX <- t(XX)
> tXX
           [,1]       [,2]       [,3]  ...         [,6]
[1,]  1.8626366  0.2091366  1.0565404  ...  0.06214715
[2,] -0.1424890  0.4177341  0.4620972  ... -0.31582345
[3,] -1.2680790 -0.3048195 -0.7531388  ...  0.37998187
           [,7]      [,8]       [,9]      [,10]
[1,] -0.7514946 0.3450489 -0.9985786 -0.9578326
[2,]  0.4713027 0.6046427 -1.5317976 -1.5308449
[3,]  0.7666866 0.6417192  1.9479734 -0.3262385
```

Compute the cross product of the two matrices (e.g., $X'X$), extract the diagonal and divide by the number of rows. If these were in deviations from the mean, the result would be the estimated variance in each column of the original matrix. Note how the division / is an element by element operation:

```
> XtX <- crossprod(XX)
> XtX
          [,1]      [,2]      [,3]
[1,]  7.944244  3.747365 -6.210131
[2,]  3.747365  7.157987 -4.951293
[3,] -6.210131 -4.951293 13.134562
> v1 <- diag(XtX)
> v1
[1]  7.944244  7.157987 13.134562
> var1 <- v1 / nrow(XX)
> var1
[1] 0.7944244 0.7157987 1.3134562
```

4.3.2 Practice

Make sure you become familiar with the difference between element by element operations and true matrix algebra in R. Practice by creating random vectors and matrices, multiply them, compute cross-products and extract diagonals. These will be important tools in constructing the test statistics needed in the simulation experiment.

4.4 Creating Spatial Weights for a Grid Layout

In many simulation settings, one controls for the spatial layout of the data by generating a sample for a regular square grid. The weights matrix for such a layout has a simple structure and does not require the checking of boundaries or computation of distances that is needed for irregular layouts. The function `cell2nb` in `spdep` creates a neighbor object for any rectangular grid, using either rook or queen as the criterion. Alternatively, a so-called torus correction may be used to avoid edge effects.[4]

The `cell2nb` function takes as required parameters the number of rows and the number of columns in the grid. In addition, the type of contiguity can be specified (default is `type="rook"`, the other option is `queen`), as well as whether or not a torus correction is needed (set `torus=TRUE`, the default is `FALSE`).

4.4.1 Example

To create a rook type neighbor object for a 4 by 4 square grid layout, and to check its properties, use:

```
> rook4x4 <- cell2nb(4,4)
> summary(rook4x4)
Neighbour list object:
Number of regions: 16
Number of nonzero links: 48
Percentage nonzero weights: 18.75
Average number of links: 3
Link number distribution:

2 3 4
4 8 4
4 least connected regions:
1:1 4:1 1:4 4:4 with 2 links
```

[4]The torus correction connects the grid cells in the right-most cell to those in the left-most cell, and those in the top row to those in the bottom row, thus ensuring that each cell has four neighbors (for rook).

```
4 most connected regions:
2:2 3:2 2:3 3:3 with 4 links
```

Compare to the properties with a torus correction:

```
> rook4x4t <- cell2nb(4,4,torus=TRUE)
> summary(rook4x4t)
Neighbour list object:
Number of regions: 16
Number of nonzero links: 64
Percentage nonzero weights: 25
Average number of links: 4
Link number distribution:

 4
16
16 least connected regions:
1:1 2:1 3:1 4:1 1:2 2:2 3:2 4:2 ... 1:4 2:4 3:4 4:4 with 4 links
16 most connected regions:
1:1 2:1 3:1 4:1 1:2 2:2 3:2 4:2 ... 1:4 2:4 3:4 4:4 with 4 links
```

4.4.2 Practice

Create neighbor objects for a 5 by 10 rectangular grid using both rook and queen as contiguity criterion, and compare their properties. Experiment with other layouts and/or a torus correction.

Creating Rectangular Grid Neighbor Objects

```
> rook5x10 <- cell2nb(5,10)
> summary(rook5x10)
Neighbour list object:
Number of regions: 50
Number of nonzero links: 170
Percentage nonzero weights: 6.8
Average number of links: 3.4
Link number distribution:

 2  3  4
 4 22 24
4 least connected regions:
```

```
1:1 5:1 1:10 5:10 with 2 links
24 most connected regions:
2:2 3:2 4:2 ... 4:9 with 4 links
> queen5x10 <- cell2nb(5,10,type="queen")
> summary(queen5x10)
Neighbour list object:
Number of regions: 50
Number of nonzero links: 314
Percentage nonzero weights: 12.56
Average number of links: 6.28
Link number distribution:

 3  5  8
 4 22 24
4 least connected regions:
1:1 5:1 1:10 5:10 with 3 links
24 most connected regions:
2:2 3:2 4:2 ... 4:9 with 8 links
```

4.5 Converting Spatial Weights to a Matrix

The neighbor list `nb` objects and the spatial weights `listw` objects both store the contiguity information in an efficient (sparse) form. Sometime, it is necessary to use the full matrix representation of the weights, for example, to use them in matrix computations.[5]

The function `nb2mat` converts the contiguity information in an `nb` object to a full matrix. A call to the `help` function shows the necessary arguments and default options:

```
> help(nb2mat)
nb2mat                 package:spdep                 R Documentation

Spatial weights matrices for neighbours lists

Description:

     The function generates a weights matrix for a neighbours list with
```

[5]Typically, however, it is *very inefficient* to operate on the full matrix, and sparse operations should be pursued as much as possible.

```
     spatial weights for the chosen coding scheme.

Usage:

     nb2mat(neighbours, glist=NULL, style="W", zero.policy=FALSE)
     listw2mat(listw)

Arguments:

neighbours: an object of class 'nb'

   glist: list of general weights corresponding to neighbours

   style: 'style' can take values W, B, C, and S

zero.policy: If FALSE stop with error for any empty neighbour sets, if
          TRUE permit the weights list to be formed with zero-length
          weights vectors

   listw: a 'listw' object from for example 'nb2listw'
```

4.5.1 Example

Convert the previously created rook neighbor list for the 4 by 4 regular grid to a full matrix, using row-standardized weights, as:

```
> wrks <- nb2mat(rook4x4)
> wrks
           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
 [1,] 0.0000000 0.5000000 0.0000000 0.0000000 0.5000000 0.0000000 0.0000000
 [2,] 0.3333333 0.0000000 0.3333333 0.0000000 0.0000000 0.3333333 0.0000000
 [3,] 0.0000000 0.3333333 0.0000000 0.3333333 0.0000000 0.0000000 0.3333333
 [4,] 0.0000000 0.0000000 0.5000000 0.0000000 0.0000000 0.0000000 0.0000000
 [5,] 0.3333333 0.0000000 0.0000000 0.0000000 0.0000000 0.3333333 0.0000000
 [6,] 0.0000000 0.2500000 0.0000000 0.0000000 0.2500000 0.0000000 0.2500000
 [7,] 0.0000000 0.0000000 0.2500000 0.0000000 0.0000000 0.2500000 0.0000000
 [8,] 0.0000000 0.0000000 0.0000000 0.3333333 0.0000000 0.0000000 0.3333333
 [9,] 0.0000000 0.0000000 0.0000000 0.0000000 0.3333333 0.0000000 0.0000000
[10,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.2500000 0.0000000
[11,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.2500000
```

```
[12,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[13,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[14,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[15,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[16,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
           [,8]      [,9]     [,10]     [,11]     [,12]     [,13]     [,14]
 [1,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [2,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [3,] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [4,] 0.5000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [5,] 0.0000000 0.3333333 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [6,] 0.0000000 0.0000000 0.2500000 0.0000000 0.0000000 0.0000000 0.0000000
 [7,] 0.2500000 0.0000000 0.0000000 0.2500000 0.0000000 0.0000000 0.0000000
 [8,] 0.0000000 0.0000000 0.0000000 0.0000000 0.3333333 0.0000000 0.0000000
 [9,] 0.0000000 0.0000000 0.3333333 0.0000000 0.0000000 0.3333333 0.0000000
[10,] 0.0000000 0.2500000 0.0000000 0.2500000 0.0000000 0.0000000 0.2500000
[11,] 0.0000000 0.0000000 0.2500000 0.0000000 0.2500000 0.0000000 0.0000000
[12,] 0.3333333 0.0000000 0.0000000 0.3333333 0.0000000 0.0000000 0.0000000
[13,] 0.0000000 0.5000000 0.0000000 0.0000000 0.0000000 0.0000000 0.5000000
[14,] 0.0000000 0.0000000 0.3333333 0.0000000 0.0000000 0.3333333 0.0000000
[15,] 0.0000000 0.0000000 0.0000000 0.3333333 0.0000000 0.0000000 0.3333333
[16,] 0.0000000 0.0000000 0.0000000 0.0000000 0.5000000 0.0000000 0.0000000
          [,15]     [,16]
 [1,] 0.0000000 0.0000000
 [2,] 0.0000000 0.0000000
 [3,] 0.0000000 0.0000000
 [4,] 0.0000000 0.0000000
 [5,] 0.0000000 0.0000000
 [6,] 0.0000000 0.0000000
 [7,] 0.0000000 0.0000000
 [8,] 0.0000000 0.0000000
 [9,] 0.0000000 0.0000000
[10,] 0.0000000 0.0000000
[11,] 0.2500000 0.0000000
[12,] 0.0000000 0.3333333
[13,] 0.0000000 0.0000000
[14,] 0.3333333 0.0000000
[15,] 0.0000000 0.3333333
[16,] 0.5000000 0.0000000
attr(,"call")
```

```
nb2mat(neighbours = rook4x4)
```

4.5.2 Practice

Create a matrix from the queen or other neighbor objects you constructed in 4.4.2. Use a matrix multiplication operation to construct a spatially lagged variable for a conforming random vector. Compare the result to what the `lag.listw` function yields (you must make sure to convert the `nb` object to a `listw` using `nb2listw`).

Creating a Spatial Lag Using Matrix Multiplication

```
> x <- rnorm(50)
> w50 <- nb2mat(queen5x10)
> w50[1,]
 [1] 0.0000000 0.3333333 0.0000000 0.0000000 0.0000000 0.3333333 0.3333333 ...
> wx <- w50 %*% x
> wwx <- lag.listw(nb2listw(queen5x10),x)
> wx[1:4]
       1:1        2:1        3:1        4:1
 0.0765760 -0.2601768  0.1325916 -0.5860503
> wwx[1:4]
[1]  0.0765760 -0.2601768  0.1325916 -0.5860503
```

4.6 A Simulation Experiment

An important property of any test statistic is its distribution under the null, and, more specifically, characteristics such as the mean, variance, and higher moments. In addition, the *size* of the test statistic is the probability of rejecting the null for a given significance level. For a test statistic to be *unbiased*, its size should equal the p-value corresponding to the theoretical critical value (the nominal significance). For example, if 1.96 is chose as the critical value in a two-sided test for a statistic that follows a standard normal distribution, 5% of the cases should reject the null hypothesis, even though it is true.

A simple simulation experiment can demonstrate these properties for Moran's I, applied to a generic random variable.[6] We will start with considering uncorrelated standard normal variates, but this can easily be extended to other distributions.

[6]The theoretical moments for Moran's I applied to regression residuals are **different**.

4.6.1 Computing Moran's I Using Matrix Algebra

For an $n \times n$ row-standardized weights matrix W with elements w_{ij}, the Moran's I statistic is:

$$I = z'Wz/z'z, \tag{4.1}$$

where z is a $n \times 1$ vector of variables expressed as deviations from the mean. When the weights are not row-standardized, a scaling factor of n/S_0 is applied, where $S_0 = \sum_i \sum_j w_{ij}$.

In our simulation experiment, we will not be relying on the built-in Moran's I calculations from `spdep`, but instead compute the statistic directly using matrix algebra and the spatial lag operation. The first step in this process is to generate a random vector and convert it to deviations from the mean:

```
> y <- rnorm(16)
> yz <- y - mean(y)
> mean(yz)
[1] -4.163336e-17
```

Next, we build the numerator and denominator of equation (4.1), using matrix products and the spatial lag operator `lag.listw` with the rook weights for the grid:

```
> yy <- crossprod(yz)
> yy
         [,1]
[1,] 13.98298
> wy <- lag.listw(nb2listw(rook4x4),yz)
> ywy <- crossprod(yz,wy)
> ywy
         [,1]
[1,] 1.294643
```

The Moran's I statistic follows directly, as:

```
> mi <- ywy / yy
> mi
           [,1]
[1,] 0.09258704
```

4.6.2 The Theoretical Moments of Moran's I

Under the normal assumption for the null, the theoretical moments of Moran's I only depend on the characteristics of the weights matrix. Specifically,

$$E[I] = \frac{-1}{n-1} \tag{4.2}$$

$$E[I^2] = \frac{n^2 S_1 - n S_2 + 3 S_0^2}{(n-1)(n+1)S_0^2}, \tag{4.3}$$

where S_1 and S_2 are sums of weights matrix elements,

$$S_1 = (1/2) \sum_i \sum_j (w_{ij} + w_{ji})^2 \tag{4.4}$$

$$S_2 = \sum_i (w_{i*} + w_{*j})^2, \tag{4.5}$$

with w_{i*} as the row sums and w_{*j} as the column sums. Note that with row-standardized weights, these expressions simplify, since each row sum equals 1, and therefore $S_0 = n$.

The variance of Moran's I then follows as

$$Var[I] = E[I^2] - E[I]^2, \tag{4.6}$$

and the statistic is turned into a standardized variate as

$$I_z = \frac{I - E[I]}{\sqrt{Var[I]}}. \tag{4.7}$$

Since the mean and variance of the statistic under the null depend solely on the weights matrix, and not on the actual variables under consideration, they must only be computed once. The required "sums" can be extracted from a spatial weights object by means of the `spdep` function `spweights.constants`.

Consider the `rook4x4` neighbor object for the square grid and extract the sums by converting the `nb` object to a `listw` object (using `nb2listw`):

```
> rks <- spweights.constants(nb2listw(rook4x4))
> rks
$n
[1] 16
$n1
[1] 15
```

```
$n2
[1] 14
$n3
[1] 13
$nn
[1] 256
$S0
[1] 16
$S1
[1] 11.05556
$S2
[1] 64.61111
```

Using these results, we can compute the moments of Moran's I in this example:

```
> eirk4x4 <- -1 / rks$n1
> eirk4x4
[1] -0.06666667
> ei2rk4x4 <- (rks$n^2 * rks$S1 - rks$n * rks$S2 + 3 * rks$S0^2)/
+ (rks$n1 * (rks$n + 1) * rks$S0^2)
> ei2rk4x4
[1] 0.03928377
> varirk4x4 <- ei2rk4x4 - eirk4x4^2
> varirk4x4
[1] 0.03483932
> sdirk4x4 <- sqrt(varirk4x4)
> sdirk4x4
[1] 0.1866530
```

4.6.3 Inference

Inference is based on a normal approximation, where the statistic is standardized and then compared to the standard normal distribution.

We standardize the statistic computed in section 4.6.1 with the moments computed above:

```
> iz <- (mi - eirk4x4)/sdirk4x4
> iz
          [,1]
[1,] 0.8532076
```

and obtain a two-tailed significance using `pnorm` with the `lower.tail = FALSE` option, and multiplying the result by 2.0

```
> piz <- pnorm(iz,lower.tail=FALSE) * 2
> piz
          [,1]
[1,] 0.3935442
```

To check this, we can use the random vector `y` in the `moran.test` with the rook grid weights (see section 3.3.1 on p. 17):

```
> moran.test(y,nb2listw(rook4x4),randomisation=FALSE,
+ alternative="two.sided")

        Moran's I test under normality

data:  y
weights: nb2listw(rook4x4)

Moran I statistic standard deviate = 0.8532, p-value = 0.3935
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
       0.09258704       -0.06666667        0.03483932
```

These results are identical to those computed "by hand."

4.6.4 Setting up the Experiment

The simulation experiment will require the following steps:

- initialize the number of simulations (`r`), sample size (`n`) and critical value (either a p-value, such as 0.05, or its counterpart in the distribution, such as 1.96)
- create a neighbor object for a given lattice layout (see `cell2nb` in section 4.4)
- convert the neighbor object to a weights list (`nb2listw`)
- compute the theoretical moments of Moran's I (section 4.6.2)
- initialize vectors with results (value of statistic, decision)

- loop over each random replication
  - generate standard normal vector (section 4.2)
  - compute Moran's I (section 4.6.1) and store result in vector
  - compute p-value (section 4.6.3) and store decision in vector
- summarize results
  - compute mean and variance (standard deviation) of results vector with Moran's I
  - do tests on normal distribution for results vector (q-q plot)
  - plot the density of the statistics, compute proportion above 1.96
  - compute percent rejection

4.6.5 The Simulation Loop

The simulation loop requires so-called "flow control." We will use a simple `for` loop, which iterates over elements in a sequence. For example, we will compute 1000 iterations of our random vectors and associated statistics.[7] We start by initializing the number of replications as well as the two results vectors:

```
> r <- 1000
> moran <- vector(mode="numeric",length=r)
> reject <- vector(mode="numeric",length=r)
```

Also, to avoid having to convert the `rook4x4` neighbor list to a `listw` object over and over again, we create a new spatial weights object, and also set the sample size to 16 and the critical value to `crit`:

```
> n <- 16
> crit <- 0.05
> w4 <- nb2listw(rook4x4)
```

Next, we organize the main computations into a loop, using `for(i in 1:r)` as the iterator:

[7]Using 1000 replications is for illustrative purposes only, a "real" simulation experiment would require a much larger sample to achieve a reasonable degree of precision.

```
> for (i in 1:r){
+     y <- rnorm(n)
+     yz <- y - mean(y)
+     yy <- crossprod(yz)
+     wy <- lag.listw(w4,yz)
+     ywy <- crossprod(yz,wy)
+     mi <- ywy / yy
+     morans[i] <- mi
+     iz <- (mi - eirk4x4)/sdirk4x4
+     piz <- pnorm(iz,lower.tail=FALSE) * 2
+     if (piz < crit) { reject[i] <- 1 }
+ }
```

Note the use of an `if` statement in the loop. The condition in parentheses is checked, and if it is evaluated as `TRUE` the action that follows is executed. In this example, this means that when the null hypothesis is rejected (as a two-sided test with a p-value of 0.05) then it is stored in the `reject` vector. Otherwise, the initialized value of zero remains.

4.6.6 Analyzing the Results

The results are contained in the vectors `morans` and `reject`. We can use the contents of `morans` to get the summary descriptive statistics for the Moran's I under the simulated null hypothesis, and use it to create histograms and density plots.

For example, the mean and variance in the simulation were (these results will obviously vary from simulation to simulation):

```
> mean(morans)
[1] -0.0744135
> sd(morans)
[1] 0.185987
> summary(morans)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.75700 -0.19800 -0.07318 -0.07441  0.04712  0.50160
```

These estimated moments can now be compared to the theoretical results obtained in section 4.6.2.

The rejection frequency is the sum of rejections (values of 1 in the vector `reject`) divided by the number of replications (`r`):

```
> rejfreq <- sum(reject) / r
> rejfreq
[1] 0.018
```

While this seems well below the nominal Type I error of `crit = 0.05`, a proper comparison needs to take into account the random variability in the simulation itself. The rejection frequency can be considered to be an estimate for the underlying probability of a series of Bernoulli random variates, each with $p = 0.05$. The variance associated with r such draws is $p(1-p)/r$. With $p = 0.05$ and $r = 1000$, this variance and the corresponding standard error of the simulation experiment are:

```
> varsim <- crit * (1.0 - crit) / r
> varsim
[1] 4.75e-05
> sdsim <- sqrt(varsim)
> sdsim
[1] 0.006892024
```

An interval around the true Type I error of two standard deviations would then be:

```
> simint <- c(crit - 2*sdsim,crit + 2*sdsim)
> simint
[1] 0.03621595 0.06378405
```

Taking this into account, our experiment would suggest that the Moran's I test statistic "significantly" under-rejects the null hypothesis in a small data set of 16 observations, using the rook criterion to define contiguity.

4.6.7 Practice

You have now all the pieces to put together a function that simulates Moran's I under the null hypothesis for a generic sample size, number of replications and weights matrix, reporting at a minimum the mean, standard deviation and rejection frequency. Using the examples in Exercise 3, you can also create density plots or histograms for the simulated statistic. You can assess the difference in the properties of the test statistic between rook and queen definition of contiguity and compare to when a torus adjustment is made. You can experiment with distributions other than the normal as well.

As a challenge, you can also consider what it would take to compute the inference under the randomization assumption.

Simulating Moran's I

For example, consider the following small function to run the simulations for any rectangular lattice, using the rook contiguity and the randomization option to carry out inference. Note that this function is *slow*, since it uses the built-in `moran.test` function (and associated overhead) in each iteration. For simulations, considerable speed up can be obtained by limiting the computations purely to the necessary items, without the overhead associated with all the features of a function like `moran.test`.

```
# sim.moran.rand.R
# generates reference distribution for Moran's I
# calculates rejection frequency using randomization
# Usage:
#    sim.moran.rand(rn,cn,rpl)
# Arguments
#    rn: number of rows in the lattice
#    cn: number of columns in the lattice
#    rpl: number of replications

sim.moran.rand <- function(rn,cn,rpl,pvalue=0.05)
{
      morans <- vector(mode="numeric",length=rpl)
      reject <- vector(mode="numeric",length=rpl)
      w <- nb2listw(cell2nb(rn,cn))
      n <- rn * cn
      for (i in 1:rpl){
           y <- rnorm(n)
           mi <- moran.test(y,w,alternative="two.sided")
           morans[i] <- mi$statistic
           if (mi$p.value) { reject[i] <- 1}
           }
      rejfreq <- sum(reject) / rpi
      value <- list(morans = morans, rej = rejfreq)
 }
```

To run this function, source it first (`>source("sim.moran.rand.R")`), then invoke it, for example. as in:

```
> resu <- sim.moran.rand(5,5,1000)
```

The result is a list containing a vector with all the computed statistics in `resu$morans` and the rejection frequency in `resu$rej`:

```
> resu$rej
[1] 0.06
```

A pdf file with a frequency plot of the simulated distribution is constructed as:

```
> pdf(file="moransim.pdf")
> plot.density(zz,main="Moran’s I under Null - Randomization",
+ xlab="Reference Distribution",lwd=2,col=2)
> dev.off()
```

The resulting plot is given in Figure 4.2 on p. 59. Other summaries are possible as well, such as histograms, qq plots or descriptive statistics.

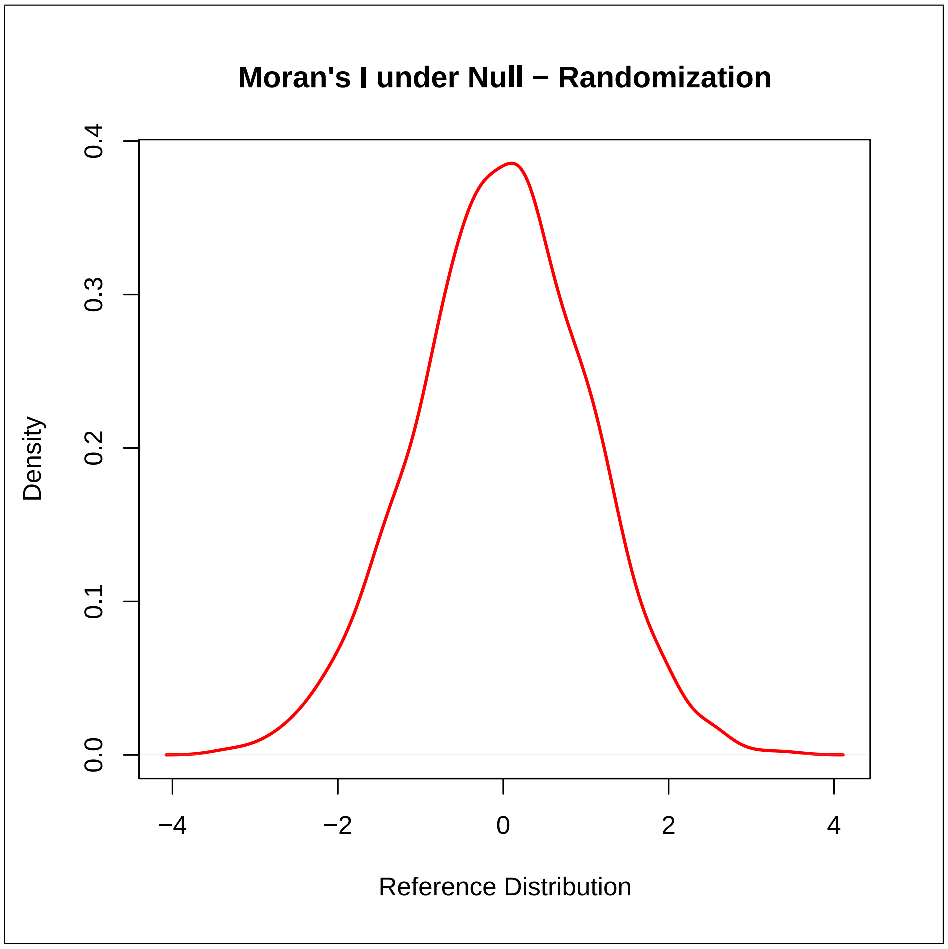

Figure 4.2: Moran's I Under the Null.

Exercise 5

Monte Carlo Simulation (2) Assessing the Properties of an Estimator

5.1 Objectives

The purpose of this exercise is to develop some further skills in designing and carrying out a Monte Carlo simulation experiment, this time focused on the properties of an estimator. The main objective is to obtain results for the bias, standard error and mean squared error of an estimator for different scenarios of error distributions. Along the way, we will review ordinary least squares regression in R, as well as how to simulate spatially correlated error terms (using `spdep`).

As before, make sure `spdep` is loaded before starting the exercises, using `library(spdep)`, and that the `columbus` data frame is attached.

5.2 Ordinary Least Squares Regression

We will focus on the ordinary least squares estimator in the linear regression model:

$$y = X\beta + \epsilon, \tag{5.1}$$

where y is a n by 1 vector of observations on the dependent variable, X is a n by k matrix of observations on the explanatory variables with matching k by 1 coefficient vector β, and ϵ is a n by 1 vector of random error terms.

For now, we will assume that the error terms are independent, identically distributed (i.i.d.).

The familiar OLS estimator for β, $\hat{\beta}$ is:

$$\hat{\beta} = (X'X)^{-1}X'y, \tag{5.2}$$

where $(X'X)$ is a k by k matrix and $X'y$ is a k by 1 vector.

Under the classical regularity conditions, the OLS estimator is unbiased, such that $E[\hat{\beta}] = \beta$ and its precision can be obtained from

$$Var[\hat{\beta}] = \sigma^2(X'X)^{-1}, \tag{5.3}$$

where σ^2 is the *unknown* error variance, estimated as $\hat{\sigma^2} = e'e/(n-k)$, with $e = y - X\hat{\beta}$ as the vector of OLS residuals.

5.2.1 OLS Using lm

The standard approach to get OLS estimates in R is to use the *linear model* or `lm` functionality. This gets passed a `formula` object, which takes the form `y ~ x1 + x2`, where `y` is the dependent variable and the variables to the right of the ~ are the explanatory variables. Check `help(lm)` for extensive documentation of the various options.

For example, using the `columbus` data frame and the variables `CRIME`, `INC`, and `HOVAL`, the standard regression yields:

```
> col1 <- lm(CRIME ~ INC + HOVAL)
> summary(col1)

Call:
lm(formula = CRIME ~ INC + HOVAL)

Residuals:
    Min      1Q  Median      3Q     Max
-34.418  -6.388  -1.580   9.052  28.649

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  68.6190     4.7355  14.490  < 2e-16 ***
INC          -1.5973     0.3341  -4.780 1.83e-05 ***
HOVAL        -0.2739     0.1032  -2.654   0.0109 *
---
```

```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 11.43 on 46 degrees of freedom
Multiple R-Squared: 0.5524,     Adjusted R-squared: 0.5329
F-statistic: 28.39 on 2 and 46 DF,  p-value: 9.34e-09
```

The result object `col1` has many attributes, including the vector of coefficients, `col1$coefficients`, the vector of residuals, `col1$residuals`, and the predicted values, `col1$fitted.values`.

5.2.2 OLS Matrix Algebra

In a simulation experiment, one needs to avoid recomputing items that do not change between replications. For example, if an OLS estimation is part of a replication loop, and the design matrix (X) does not change between replications, using a call to `lm` is not a good idea. Instead, you want to compute all the parts that do not change *before* the loop is started and only keep what changes inside the loop.

In the case of OLS, this means that we need to compute the estimates by using matrix algebra. From equation 5.2, we know that this involves the computation of two cross products, the inverse of a cross product, and the product of a matrix with a vector. The matrix products are straightforward to implement, using the `crossprod` and `%*%` operators. The inverse is something that usually requires some care, and for numerical stability a Cholesky decomposition or qr approach is often preferable. For this example, we will use the `solve` operator. This supports two operations, the inverse of a matrix, such as `solve(A)`, and the *solution* to a system of linear equations, as `solve(A,b)`, where the result equals $A^{-1}b$.

5.2.3 Example

Before proceeding to the solution of the OLS normal equations, we first need to turn the relevant elements of the `columbus` data frame into vectors and matrices. For the dependent variable, this is a simple assignment:

```
> y <- CRIME
> y
 [1] 15.725980 18.801754 30.626781 ...  0.178269
 [8] 38.425858 30.515917 34.000835 ... 57.066132
...
[43] 36.663612 25.962263 29.028488 ... 22.541491
```

For the matrix of observations on the explanatory variables, X, we use the `cbind` function and exploit the recycling rule to add a vector of ones as the first column:

```
> X <- cbind(1,INC,HOVAL)
> X
          INC  HOVAL
 [1,] 1 19.531 80.467
 [2,] 1 21.232 44.567
...
[49,] 1 18.796 35.800
```

Note how even though we only include the *scalar* 1, the recycling rule turns this into a vector of the same length as the other two vectors.

Next, we need to construct the matrix of cross products $X'X$ and the vector $X'y$:

```
> XX <- crossprod(X)
> XX
                    INC     HOVAL
        49.000   704.372  1883.375
INC    704.372 11686.673 29600.442
HOVAL 1883.375 29600.442 88757.619
> Xy <- crossprod(X,y)
> Xy
          [,1]
      1721.312
INC   21557.532
HOVAL 57640.624
```

Note how we only need to specify `X` as the argument in the first `crossprod` operation, since R constructs $X'X$ by default. The inverse $(X'X)^{-1}$ follows directly, as:

```
> XXi <- solve(XX)
> XXi
                              INC         HOVAL
       0.171498009 -0.0072068055 -0.0012356172
INC   -0.007206805  0.0008538132 -0.0001318211
HOVAL -0.001235617 -0.0001318211  0.0000814476
> XX %*% XXi
```

```
                                 INC         HOVAL
        1.000000e+00  3.018903e-17  7.543415e-18
  INC   -5.933489e-15  1.000000e+00 -1.250043e-16
  HOVAL -1.711690e-14 -1.422552e-15  1.000000e+00
```

The OLS estimates follow as:

```
> bols <- XXi %*% Xy
> bols
            [,1]
      68.6189611
INC   -1.5973108
HOVAL -0.2739315
```

A more efficient (and numerically more accurate) way is to include both `XX` and `Xy` in the `solve` operation, as:

```
> b <- solve(XX,Xy)
> b
            [,1]
      68.6189611
INC   -1.5973108
HOVAL -0.2739315
```

The drawback of this for our simulation is that it does not give us a separate $(X'X)^{-1}$, which is the part that remains constant between replications. Also, we need the inverse separately if we want to compute the standard errors of the OLS estimates and the usual t-test statistics.

First, we need to obtain an estimate for the error variance. This requires the calculation of the residuals and their sum of squares, which then need to be divided by the degrees of freedom, as:

```
> e <- y - X %*% b
> e

               [,1]
   [1,]   0.3465419
   [2,]  -3.6947990
...
  [49,]  -6.2476690
> ee <- crossprod(e)
> ee
```

```
          [,1]
  [1,] 6014.893
> vare <- ee / (nrow(X) - ncol(X))
> vare

          [,1]
  [1,] 130.7585
  > sqrt(vare)

          [,1]
  [1,] 11.43497
```

Compare to the results printed in the summary of the `lm` function.

The standard errors of the estimates are the square roots of the diagonal elements of the matrix $\hat{\sigma^2}(X'X)^{-1}$. We extract these by means of the `diag` function and apply the square root function, to yield:

```
> varb <- vare * diag(XXi)
> seb <- sqrt(varb)
> seb
[1] 4.7354861 0.3341308 0.1031987
```

Note how we do not need to use the matrix multiplication operator to multiply the scalar `vare` with the diagonal elements of the inverse matrix. The t-test statistics then follow as the ratio of the coefficient estimates over their standard error:

```
> tb <- b / seb
> tb
          [,1]
     14.490373
INC   -4.780496
HOVAL -2.654409
```

The associated probabilities are found from the Student t distribution (note the use of the `abs` function for this to work properly):

```
> pb <- pt(abs(tb),df=nrow(X)-ncol(X),lower.tail=FALSE) * 2
> pb
             [,1]
      9.210890e-19
```

```
INC   1.828960e-05
HOVAL 1.087450e-02
```

Again, compare to the output of `lm` applied to the Columbus example.

5.2.4 Practice

Use the POLICE sample data set to replicate the regression in the 1992 *Regional Science and Urban Economics* Kelejian-Robinson article. For simplicity, you could also use a subset of the regressors. The Kelejian-Robinson specification was a regression of police expenditures on the tax rate, intergovernmental transfers, per capital income, crime rate, unemployment rate, home ownership, percent college, percent white and percent commuters (see Kelejian and Robinson 1992, pp. 323–324). Use both the standard `lm` function as well as an explicit matrix computation.

Kelejian-Robinson Baseline OLS Regression

```
> police.ols <- lm(POLICE ~ TAX + TRANSFER + INC + CRIME +
+ UNEMP + OWN + COLLEGE + WHITE + COMMUTE)
> summary(police.ols)

Call:
lm(formula = POLICE ~ TAX + TRANSFER + INC + CRIME + UNEMP +
    OWN + COLLEGE + WHITE + COMMUTE)

Residuals:
    Min      1Q  Median      3Q     Max
-841.75  -99.18   15.94  127.21 1007.11

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -4.858e+02  5.136e+02  -0.946   0.3474
TAX          2.182e-01  4.715e-01   0.463   0.6450
TRANSFER     7.553e-02  2.819e-03  26.789  < 2e-16 ***
INC          1.028e-01  5.031e-02   2.043   0.0447 *
CRIME        1.331e+00  1.661e-01   8.011 1.46e-11 ***
UNEMP       -1.990e+01  1.724e+01  -1.154   0.2522
OWN         -7.929e+00  5.658e+00  -1.401   0.1654
COLLEGE     -3.686e-01  4.013e+00  -0.092   0.9271
WHITE       -3.408e-01  2.586e+00  -0.132   0.8955
```

```
COMMUTE      3.268e+00  2.996e+00   1.091   0.2789
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 272.4 on 72 degrees of freedom
Multiple R-Squared: 0.9708, Adjusted R-squared: 0.9671
F-statistic: 265.9 on 9 and 72 DF,  p-value: < 2.2e-16
```

5.3 Spatial Autoregressive Random Variables

In Exercise 4, we conducted a simulation experiment under the null, when it was not necessary to specify a particular form or magnitude for the spatial autocorrelation. In order to obtain spatially autocorrelated random variables, we will need to apply a transformation that turns uncorrelated variates into correlated ones. We will focus on spatial autoregressive (SAR) and spatial moving average (SMA) processes, although this by no means exhausts the set of possibilities. In this section, we deal with SAR processes. SMA processes are treated in section 5.4.

In its simplest form, a spatial autoregressive process is:

$$\epsilon = \rho W \epsilon + u \tag{5.4}$$

where u and ϵ are n by 1 random vectors, ρ is the autoregressive parameter, and W is a n by n spatial weights matrix. The u vector is i.i.d., and ϵ is spatially correlated. We can obtain ϵ directly from u by means of a so-called spatial autoregressive transformation:

$$\epsilon = (I - \rho W)^{-1} u. \tag{5.5}$$

The package `spdep` contains the function `invIrM` which constructs the inverse matrix $(I - \rho W)^{-1}$ required in a spatial autoregressive transformation. It takes as arguments a neighbor list (`nb`) or weights list (`listw`) object, followed by a value for the autoregressive parameter (`rho`). In addition, a list of general weights can be optionally included, as well as a specific style for the weights (the default is `style="W"`, a row-standardized weights matrix). The result is an n by n matrix. In contrast to the original spatial weights, which tend to be sparse, the inverse is typically dense, requiring a lot of memory for larger data sets.

5.3.1 Example

Use the `col.gal.nb` neighbor list for Columbus to create a spatial autoregressive transformation matrix, with 0.5 for the autoregressive parameter:

```
> c05 <- invIrM(col.gal.nb,0.5)
```

Now use this transformation matrix to turn a vector of 49 standard (uncorrelated) random variates into a vector of spatially correlated random variables, as:[1]

```
> uu <- rnorm(49)
> mean(uu);sd(uu)
[1] 0.2138447
[1] 1.247263
> e05 <- c05 %*% uu
> mean(e05);sd(e05)
[1] 0.3113457
[1] 1.397368
```

Check with a Moran spatial autocorrelation test:

```
> moran.test(e05,nb2listw(col.gal.nb),randomisation=FALSE,
+ alternative="two.sided")

        Moran's I test under normality

data:  e05
weights: nb2listw(col.gal.nb)

Moran I statistic standard deviate = 3.4981, p-value = 0.0004687
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.308448612      -0.020833333       0.008860962
```

5.3.2 Practice

Use one of the spatial weights created for the regular lattices in Exercise 4, or the spatial weights for the POLICE sample data set to generate vectors

[1]While the standard deviation of the spatially correlated `e05` is reported, this is not totally valid, since these variables are heteroskedastic and thus do not have a common variance.

with spatially autocorrelated normal random variables. Experiment with different values for the spatial autoregressive parameter and check if the `moran.test` rejects the null. Specifically, check for small values of `rho`, such as 0.1 or 0.05.

Creating Spatially Autoregressive Variables

Consider a simple function (without any error checking) that takes as arguments a vector, a `nb` object and a value for the autoregressive parameter. It uses this and the `spdep` function `invIrW` to create a spatial autoregressive transformation.

```
# sar.transform
# transform a vector into spatially autoregressive vector
# Usage:
#    sar.transform(x,w,rho)
# Arguments:
#    x: vector of values
#    w: spatial weights nb object
#    rho: spatial autoregressive coefficient
# Value:
#    vector
# uses spdep invIrM
# WARNING: no error checking!

sar.transform <- function(x,w,rho)
{
    irw <- invIrM(w,rho)
    wx <- irw %*% x
    wx
}
```

Create a spatial autoregressive transformation of the `POLICE` variable, and test for spatial autocorrelation as:

```
> source("sar.transform.R")
> wpolice05 <- sar.transform(POLICE,polgal,0.5)
> moran.test(wpolice05,polrook,alternative="two.sided")

Moran's I test under randomisation
```

```
data:  wpolice05
weights: polrook

Moran I statistic standard deviate = 6.1307, p-value = 8.752e-10
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.365968180      -0.012345679       0.003807943
```

5.4 Spatial Moving Average Random Variables

A spatial moving average process is:

$$\epsilon = \rho W u + u. \tag{5.6}$$

In order to construct a random variable that follows this process, no inverse is required, but only the sum of a random variable and its spatial lag, the latter multiplied by the spatial parameter. This is a simple operation that does not require matrix operations, but can be implemented using the sparse `lag.listw` function in spdep.

5.4.1 Example

Using the same `col.gal.nb` neighbor list and the same vector of standard normal variates `uu` as in Section 5.3.1, create a spatial lag for the random vector:

```
> wu <- lag.listw(nb2listw(col.gal.nb),uu)
```

Now, carry out a spatial moving average transformation as:

```
> em05 <- 0.5 * wu + uu
> mean(em05);sd(em05)
[1] 0.2490343
[1] 1.279354
```

Compare the degree of spatial autocorrelation to that of the SAR counterpart with a `moran.test`:

```
> moran.test(em05,nb2listw(col.gal.nb),randomisation=FALSE,
+ alternative="two.sided")
```

```
        Moran's I test under normality

data:  em05
weights: nb2listw(col.gal.nb)

Moran I statistic standard deviate = 2.5339, p-value = 0.01128
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.217686051      -0.020833333       0.008860962
```

5.4.2 Practice

Use the same weights and normal random variates as in 5.3.2 to create a new random vector that follows a spatial moving average process. Compare the degree of spatial autocorrelation as indicated by a Moran test for different values of the spatial parameter.

Creating Spatial Moving Average Variables

Again, consider a simple function (without any error checking) that takes as arguments a vector, a `nb` object and a value for the spatial moving average parameter. It uses this and the `spdep` function `lag.listw` to create a spatial moving average transformation.

```
# sma.transform
# transform a vector into spatial moving average vector
# Usage:
#    sma.transform(x,w,rho)
# Arguments:
#    x: vector of values
#    w: spatial weights nb object
#    rho: spatial autoregressive coefficient
# Value:
#    vector
# uses spdep lag.listw
# WARNING: no error checking!

sma.transform <- function(x,w,rho)
{
    wx <- lag.listw(nb2listw(w),x)
    smax <- rho * wx + x
    smax
}
```

As before, source the function and invoke it to transform the `POLICE` variable into a moving average form.

```
> source("sma.transform.R")
> smapolice05 <- sma.transform(POLICE,polgal,0.5)
> moran.test(smapolice05,polrook,alternative="two.sided")

Moran's I test under randomisation

data:  smapolice05
weights: polrook

Moran I statistic standard deviate = 5.1891, p-value = 2.113e-07
alternative hypothesis: two.sided
sample estimates:
Moran I statistic       Expectation          Variance
      0.302526841      -0.012345679       0.003682022
```

5.5 A Simulation Experiment

We will implement a simple simulation experiment to assess the properties of the OLS estimator when the regression error term follows a spatial autoregressive process. We know from theory that OLS should remain unbiased, but will be less efficient than for i.i.d. errors. Also, the standard errors of the estimates as suggested by OLS will be biased and tend to underestimate the true variability of the estimates.

The basic design of the experiment consists of generating vectors of simulated "observations" on y by using $y = X\beta + \epsilon$, for a fixed matrix X and a given parameter vector β. The ϵ vectors are constructed by a spatial autoregressive transformation. Starting with an i.i.d. random vector u, a value for the parameter ρ and a spatial weights matrix, $\epsilon = (I - \rho W)^{-1}u$. The simulated ϵ is added to $X\beta$ to yield the y vector. The product $X\beta$ is fixed and must only be computed once, so it should be *outside* the main simulation loop.

There are many design aspects that need to be controlled for in a simulation experiment such as this one. Paradoxically, one aspect that requires very little attention is the choice of the value for the *true* β parameters. Typically, one takes a single vector of random X values and sets $\beta = 1$ for simplicity. In order to make sure the regressions have a similar and reasonable fit, the variance of the simulated X is chosen together with the variance of the error term. Consider that the true regression total variance (of y) can be decomposed into variance due to X and variance due to the error term, we select the two variances such that the regression R^2 is something like 0.9. For example, we could randomly generate X as a normal variate with mean zero and variance 9, and the error term as a standard normal variate. This would yield regressions with "on average" an R^2 of 0.9. Note that this only holds under the null of no spatial autocorrelation. The spatially correlated error terms will no longer be homoskedastic, so that the notion of an R^2 is not valid in a strict sense, nor is there a common error variance.

The properties of the estimator can be quantified in a number of ways. First, we consider the *bias* of the estimator, which is estimated as the difference between the average of $\hat{\beta}$ across all replications and the true parameter value β:

$$bias = \sum_r \hat{\beta}_r / R - \beta \tag{5.7}$$

where R is the total number of replications. The precision of the estimator can be measured as the variance (or, standard error) of the reference distribution of the $\hat{\beta}_r$:

$$Var[\hat{\beta}] = (1/R) \sum_r (\hat{\beta}_r - \bar{\hat{\beta}})^2, \tag{5.8}$$

where $\bar{\hat{\beta}} = \sum_r \hat{\beta}_r / R$, and the standard error is the square root of this result. The result for the simulated data can be compared to its theoretical counterpart, the matching diagonal element in $\sigma^2(X'X)^{-1}$ with σ^2 replaced by the value chosen for the simulated error term.

Alternatively, the measure of precision can be computed with respect to the *true* value of β, yielding the *mean squared error* (MSE):

$$MSE[\hat{\beta}] = (1/R) \sum_r (\hat{\beta}_r - \beta)^2. \tag{5.9}$$

The MSE is particularly appropriate when comparing estimators that are not necessarily unbiased, since it consists of the sum of the variance and the squared bias. For an unbiased estimator, the latter is zero ("in expected value") and MSE and variance are the same.

5.5.1 Setting up the Experiment

The simulation experiment will require the following steps:

- initialization of parameters
  - set the number of replications (`r`), sample size (`n`), value range for the autoregressive parameters (`rho`), variance of the error terms (`sigma`)
- initialization and preprocessing of X matrix
  - generate a random vector (`x`) with variance such that the regression R^2 is a given target value (e.g., 0.9).
  - create the `X` matrix as 1 + `x`, this implicitly sets β to (1.0, 1.0)
  - compute `XXi` $= (X'X)^{-1}$
  - compute `vb2` $= \sigma^2(X'X)^{-1}_{2,2}$, the theoretical variance of OLS
- initialization and preprocessing of spatial weights
  - create a neighbor object for a chosen regular lattice layout (`cell2nb`)
  - if needed, convert neighbor object to list weights (`nb2listw`)
- initialize vector (or matrix) to hold estimated coefficients
- loop over values for `rho`
  - create SAR transformation matrix for given `rho` and weights (`invIrM`)
- main simulation loop
  - generate i.i.d. random vector `uu`
  - transform `uu` to spatially autoregressive vector `eps`
  - compute dependent variable vector `y` = `Xb` + `eps`
  - compute cross product `Xy` $= X'y$
  - obtain OLS estimate `b` = `XXi %*% Xy`
  - store `b` in results vector
- summarize results
  - compute average `b`

– compute bias

– compute variance of `b`

– compare variance to "theoretical" variance

– compute MSE of `b`

– (optional) graph reference distribution of `b`

5.5.2 Example

We start with the various initializations, to keep things generic for later incorporation into a function:

```
> n <- 49
> r <- 1000
> rho <- c(0.0, 0.2, 0.5, 0.7, 0.9)
```

Next we construct the `X` matrix, its cross product and inverse:

```
> X <- cbind(1,rnorm(n,0,3))
> XX <- crossprod(X)
> XX
          [,1]       [,2]
[1,] 49.000000  -3.904727
[2,] -3.904727 391.827489
> XXi <- solve(XX)
> XXi
             [,1]         [,2]
[1,] 0.0204243828 0.0002035376
[2,] 0.0002035376 0.0025541719
> vb2 <- XXi[2,2]
> vb2
[1] 0.002554172
```

Note that we use 3 as the standard deviation in the standard normal variate (to obtain a variance of 9) and how again the recycling rule is exploited to create the constant term. The theoretical variance of the OLS estimator for β_2 is extracted as the `[2,2]` element of the variance matrix (since $\sigma^2 = 1$). Also, the value of $X\beta$ can be pre-computed as the row sum of the matrix `X` (rather than an explicit matrix-vector multiplication):

```
> Xb <- rowSums(X)
```

Now, we construct a neighbor list for rook contiguity on a 7 by 7 regular lattice. The spatial autoregressive transformation is different for each value of `rho`, so that must be carried out inside the loop over the values of `rho`:

```
> w <- cell2nb(7,7)
```

Next, initialize the matrix that will hold the results for β_2 (the slope) for each of the values of `rho` (0.0 to 0.9, as columns) and in each of the replications:

```
> bres <- matrix(0, nrow=r, ncol= length(rho))
> bres
        [,1] [,2] [,3] [,4] [,5]
   [1,]    0    0    0    0    0
...
 [999,]    0    0    0    0    0
[1000,]    0    0    0    0    0
```

We are now ready for the main simulation loop. There is actually a double loop, one over the elements of `rho`, the second over the replications. In the outer loop, the spatial autoregressive transformation matrix is built for the corresponding value of `rho`. In the inner loop, the random errors are generated and transformed, the "observed" `y` is constructed, the regression is estimated and the result for $\hat{\beta}$ stored in `bres`:

```
> for (j in 1:length(rho)) {
+        iw <- invIrM(w,rho[j])
+        for (i in 1:r) {
+            e <- iw %*% rnorm(n)
+            y <- Xb + e
+            Xy <- crossprod(X,y)
+            b <- XXi %*% Xy
+            bres[i,j] <- b[2]
+        }
+ }
```

The summary characteristics of the OLS estimator are:

```
> summary(bres)
       X1                X2                X3                X4
 Min.   :0.8474   Min.   :0.8357   Min.   :0.8364   Min.   :0.8155
 1st Qu.:0.9670   1st Qu.:0.9663   1st Qu.:0.9640   1st Qu.:0.9637
 Median :0.9996   Median :1.0015   Median :0.9986   Median :1.0013
 Mean   :0.9993   Mean   :1.0006   Mean   :0.9999   Mean   :1.0006
 3rd Qu.:1.0333   3rd Qu.:1.0361   3rd Qu.:1.0349   3rd Qu.:1.0392
 Max.   :1.1515   Max.   :1.1707   Max.   :1.2232   Max.   :1.1838
       X5
 Min.   :0.7475
 1st Qu.:0.9337
 Median :0.9990
 Mean   :0.9985
 3rd Qu.:1.0607
 Max.   :1.2515
```

This illustrates how the mean and median don't change much from the expected value of 1, but the range of the distribution widens with ρ (first slightly, but then quite dramatically with $\rho = 0.9$). More precise characteristics of the distribution are the average (`bavg`, note the use of the function `colMeans`), the bias (`bias`), variance (`varb`, note again the use of `colMeans`), the standard error (`sdeb`), difference with the theoretical standard error (the square root of `vb2`), and the mean squared error (`mseb`):

```
> bavg <- colMeans(bres)
> bavg
[1] 0.9993119 1.0006317 0.9999488 1.0006297 0.9985044
> bias <- bavg - 1.0
> bias
[1] -6.880809e-04  6.316951e-04 -5.124518e-05  6.296576e-04
    -1.495599e-03
> varb <- colMeans((bres - bavg)^2)
> varb
[1] 0.002373828 0.002616419 0.002882452 0.003484847 0.008237334
> sdeb <- sqrt(varb)
> sdeb
[1] 0.04872195 0.05115094 0.05368848 0.05903260 0.09075976
> sdeb - sqrt(vb2)
[1] -0.0018168670  0.0006121246  0.0031496605  0.0084937801
     0.0402209418
> mseb <- colMeans((bres - 1)^2)
> mseb
[1] 0.002371185 0.002614124 0.002878571 0.003485413 0.008236709
```

In essence, the OLS estimator remains unbiased, even with $\rho = 0.9$, but its variance, standard error and mean squared error are affected. They increase with ρ, first slightly, but then much more so for $\rho = 0.9$.

To end in beauty, we will plot the empirical densities for the estimates obtained in the simulation, for each of the values of ρ and use some fancy plotting techniques to combine them all in one graph, plot in different colors, and add a legend to boot. As in Exercise 3, we start by turning the columns of the matrix `bres` into `density` objects, `zz0` through `zz4`:

```
> zz0 <- density(bres[,1])
> zz1 <- density(bres[,2])
> zz2 <- density(bres[,3])
> zz3 <- density(bres[,4])
> zz4 <- density(bres[,5])
```

Next, we draw a density plot for the last graph (`zz4`), set the `main` title and the label for the x-axis (`xlab`), and specify the color to be red (2). In addition, to make sure that all the plots will fit on the same graph, we use the min and max values from the `summary(bres)` to set the `xlim` and `ylim` parameters (the range for the values on the x- and y-axis respectively). We further add the other graphs as `lines` (since the `add` option is not implemented for a `density.plot`), specifying the x-value as `zz $x` and the y-value as `zz $y`, with a different color for each. Finally, we add a `legend` (one of the more complex but also more powerful options). We set the coordinates for the upper left corner, specify the text for the legend (using `paste` to add the words `rho = ` in front of the value taken from the `rho` vector), set the type to line (`lty=1`), in the same dimension as the legend "words" (i.e., a five element vector), and set the matching colors in a vector (`col`):

```
> plot.density(zz4,main="Density of beta hat",xlab="beta hat",
+ xlim=c(0.7,1.3),ylim=c(0,8),col=2)
> lines(zz3$x,zz3$y,col=6)
> lines(zz2$x,zz2$y,col=5)
> lines(zz1$x,zz1$y,col=3)
> lines(zz0$x,zz0$y,col=4)
> legend(0.7,8,paste("rho = ",rho),lty=c(1,1,1,1,1),
+ col=c(4,3,5,6,2))
```

The result looks like Figure 5.1, illustrating the effect of ρ on the spread of the distribution (of course, you need colors to distinguish the lines).

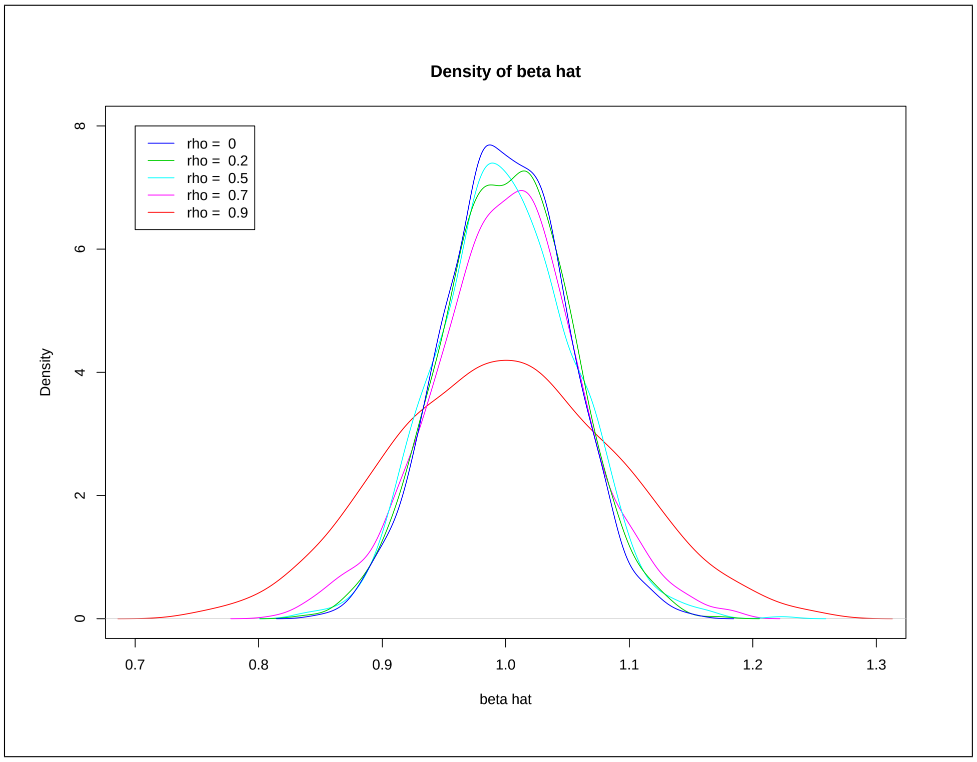

Figure 5.1: Empirical Distribution of $\hat{\beta}$ for different ρ.

5.5.3 Practice

You can now combine the various commands into a function that allows you to make the sample size generic, specify the type of distribution for the error term, include both spatial autoregressive as well as spatial moving average transformations, etc. In addition, you could try to assess the effect on OLS of an omitted spatial lag variable (the transformation is slightly more complex than in the error case). The sky is the limit ...

Exercise 6

Spatial Regression in R

6.1 Objectives

This exercise is an overview of specification testing for spatial autocorrelation in regression residuals and the estimation of spatial regression models by means of Maximum Likelihood. These materials are contained in the *Regression* tutorial (Anselin 2003b, specifically, pp. 5–6 and 10–17). You are referred to that tutorial for specific instructions and examples.

6.2 Things You Should Know

6.2.1 Running a Regression

- using `lm` with a "formula," `y ~ x1 + x2`
- assigning the regression results to an object
- extracting the residuals as `resid(regressionObject)`

6.2.2 Testing OLS Residuals for Spatial Autocorrelation

- Moran test for residuals using `lm.morantest`
- parameters: a regression object and a `listw` object
- Lagrange Multiplier tests using `lm.LMtests`
- parameters: a regression object, `listw` object and the type of test

- specifying the LM tests by name (`LMerr`, `RLMerr`, `LMlag`, `RLMlag`, `SARMA`)

6.2.3 ML Estimation of Spatial Lag Model

- using `lagsarlm` with a formula and `listw` object
- using `summary` to list the detailed results

6.2.4 ML Estimation of Spatial Error Model

- using `errorsarlm` with a formula and `listw` object
- using `summary` to list the detailed results

6.2.5 Testing the Spatial Common Factor Hypothesis

- using `lagsarlm` with `type="mixed"`
- using `LR.sarlm` for the Likelihood Ratio test

6.3 Practice

Use the Boston sample data and model specification to run through a spatial specification search and model estimation. Alternatively, experiment with one of the data sets in the SAL collection of sample data sets. You will need to first convert the shape files to R data frames and construct spatial weights. Optionally, compare the results between R and *GeoDa*.

Appendix

Summary of Data and Spatial Weights Operations

- Data import and setup
  - `library(spdep)`: load the `spdep` package library
  - `source("read.geoda.R")`: load and compile the source file *read.geoda.R*
  - `read.geoda`: read a data set exported from *GeoDa*
  - `data(dataset)`: load an R data set
  - `attach(dataset)`: attach the data frame, making the variables available by name
  - `summary(dataset)`: summary descriptive statistics of all variables in the data set
- Types of spatial weights objects
  - `nb`: neighbor list, contiguities only
  - `listw`: neighbor list with weights, sparse format
  - `knn`: k-nearest neighbor object
  - `matrix`: standard matrix object
- Reading spatial weights files into neighbor lists
  - `read.gal`: read GAL format weights files
  - `read.gwt2nb`: read GWT format weights files
- Creating spatial weights objects
  - `cell2nb(nrow,ncol)`: neighbor list for rectangular regular grid cells

- `dnearneigh(coordinatematrix,lowdist,updist)`: create distance band weights using the `lowdist` and `updist` limits
- `knearneigh(coordinatematrix,k)`: create a matrix with the indices of the k-nearest neighbors
- `nblag(nb,maxlag)`: higher order neighbors
- `poly2nb(polygonlist)`: neighbor object from polygon boundary file
- `tri2nb(coordinatematrix)`: create neighbor list using Thiessen polygon neighbor structure

- Conversion between weights objects
  - `knn2nb(knn)`: create a neighbor list object from a `knn` object (a matrix)
  - `nb2listw(nb)`: create a `listw` weights object from a neighbor list
  - `mat2listw(matrix)`: convert a spatial weights n by n matrix to a `listw` object
  - `nb2mat(nb)`: convert a neighbor list to a matrix
  - `make.sym.nb(nb)`: force a possibly asymmetric neighbor list (e.g., from k-nearest neighbors) to a symmetric form
- Characteristics of spatial weights
  - `summary(nb)`: connectivity summary of neighbor list
  - `card(nb)`: number of neighbors for each observation
  - `is.symmetric.nb(nb)`: check on symmerty of connectivity structure
- Spatial transformations
  - `lag.listw(listw,variables)`: create spatially lagged variables using a `listw` object
  - `invIrM(listw,rho)`: create inverse matrix $(I - \rho W)^{-1}$

Bibliography

Anselin, L. (2003a). *Data and Spatial Weights in spdep, Notes and Illustrations.* http://sal.agecon.uiuc.edu/stuff_main.php#tutorials.

Anselin, L. (2003b). *An Introduction to Spatial Regression Analysis in R.* http://sal.agecon.uiuc.edu/stuff_main.php#tutorials.

Bivand, R. S. (2001). More on spatial data analysis. *R News*, 1(3):13–17.

Bivand, R. S. (2002a). Implementing spatial data analysis software tools in R. In Anselin, L. and Rey, S., editors, *New Tools for Spatial Data Analysis: Proceedings of the Specialist Meeting.* Center for Spatially Integrated Social Science (CSISS), University of California, Santa Barbara. CD-ROM.

Bivand, R. S. (2002b). Spatial econometrics functions in R: Classes and methods. *Journal of Geographical Systems*, 4:405–421.

Bivand, R. S. and Gebhardt, A. (2000). Implementing functions for spatial statistical analysis using the R language. *Journal of Geographical Systems*, 2:307–317.

Bivand, R. S. and Portnov, B. A. (2004). Exploring spatial data analysis techniques using R: The case of observations with no neighbors. In Anselin, L., Florax, R. J., and Rey, S. J., editors, *Advances in Spatial Econometrics: Methodology, Tools and Applications*, pages 121–142. Springer-Verlag, Berlin.

Kelejian, H. H. and Robinson, D. P. (1992). Spatial autocorrelation: A new computationally simple test with an application to per capita country police expenditures. *Regional Science and Urban Economics*, 22:317–333.

Venables, W., Smith, D., and R Development Core Team (2004). *An Introduction to R – Notes on R, a Programming Environment for Data Analysis and Graphics, Version 2.0.1 (2004-11-15).* http://cran.r-project.org.