Spatial Analysis Techniques in R

Using Spatial Data in R

Lesson 1: Introducing R-Spatial and displaying some geographic data

1.1 Introduction

The R project is described in the website *http://www.r-project.org*, and R is available as free software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS. R is one of the most successful open source projects, with the result that it is extremely well-supported by a comprehensive manual, almost 100 texts that make use of it, a wiki, an on-line journal and annual user conferences. There are a lot of these resources at the R website. However, R is NOT primarily a geographical information system (GIS) or, for that matter, a spatial statistical analysis system (such as *GeoDa* ™ or *CRIMESTAT* III) nor is it a mapping program such as *3DField* or *SURFER*™. That said R can be used as a GIS if you know how and its ability to combine standard statistical, spatial statistical and geometric operations in the same consistent environment is a considerable advantage.

R is really a computing *environment*, sometimes called an *ecosystem*, a suite of software facilities for data manipulation, calculation and graphical display that includes:

- An effective data handling and storage facility;
- A suite of operators for calculations on arrays, in particular matrices;
- A large, coherent, integrated collection of intermediate tools for data analysis;
- Graphical facilities for data analysis and display either on-screen or on hardcopy; and
- A well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term *environment* or *ecosystem* is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of specific and inflexible tools, as is frequently the case with other data analysis software and it makes sense to think of it as an environment within which statistical techniques can be implemented rather easily. As we will see, R can be extended (easily) via contributed *packages*. There are several packages supplied with the base R distribution and many more are available, including GIS-like ones, through the CRAN family of Internet sites.

There are many reasons why a spatial analyst might chose to use R, such as:

- It is easy using R to move data to and from statistical packages;
- Many packages in R have interfaces to GIS, which means it is also easy to move data backwards and forwards as necessary;
- In R there are implementations of new methods either not yet in or are difficult to use in standard GIS;

- Working in the R environment enables rapid development of code for new methods, and it has become the environment of choice for most spatial statisticians so there is an active community to whom you can turn for help;
- Many spatial operations have relatively high order computational complexity, which means that any code has to be efficient. By and large it will be found that such operations run more efficiently in R than they do in a GIS.

As we will see perhaps the main 'downsides' for a spatial analyst working in R are:

- *Data structures*: by and large statisticians like to work with simple tables of numbers in which rows represent cases and columns any variables attached to those cases (in R and its predecessor S, this is the data.frame) and the result is a simple matrix of numbers (or characters). A moment's thought will indicate that if we are to analyze and use geographical location we need to be able to recognize and handle more complex data structures than this. Fortunately, there are packages and associated data classes in R to help us do this;
- *Unfamiliarity*: which means the 'buy in price' can seem quite high, with a steep learning curve. For many, working in a rigorous 'object' framework will take a while to get used to. Others will find working with a command line interface rather than 'pointing and clicking' on drop down menus awkward and for others the case sensitivity of the commands can give rise to errors that in practice can be quite hard to see;
- *Choice:* Although the core of R was planned there is a huge range of available packages, which vary in their efficiency and utility. Specifically, the range of available packages for spatial analysis is wide and it is often not clear which might be best for a particular problem.

Despite all of this, I am not myself sure that the learning curve is all that steeper than it would be for a conventional GIS such as ArcGIS or QGIS, or for a series of stand-alone programs such as CRIMESTAT, *GeoDa*™ and *3Dfield* used in *statistics.com*'s introductory course in spatial analysis. The gains seem to me to be well-worth the effort.

If you have a copy Brunsdon and Comber (2015) Chapter 1, pages 1-9 covers this in some detail but reading it is not essential to this course.

Task 1.1: Getting R

If you have already got a fairly recent version of R on your machine as far as I can see everything should work OK, at least with a 32-bit build.

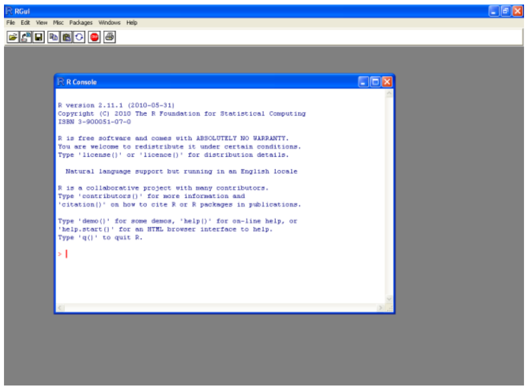
- You may already have R implemented on your machine, but if not, follow the link from *www.r-project.org* and visit a *Comprehensive R Archive Network* (=”CRAN”) ‘mirror’ site that will facilitate the download;

- Download the most recent version, which will be R 3.2.3 at the time of writing. It’s quite a big, (62Mb) so the download might need patience). This is a pre-compiled binary distribution of what’s called the base system (you can also download the source), with choice of operating system between Linux, MacOS and various flavors of Windows;

- If you run Windows 7 64 bit please read Sections 2.24 and 2.27 of @R for Windows FAQ’ and please follow the advice to use the 32-bit build:

- RUN (open) the relevant .exec file and accept all the defaults.

Once installed, click on the R icon to get a screen that will look something like that below:



```
R version 2.11.1 (2010-05-31)
Copyright (C) 2010 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

If for any reason you can’t get to this point then please let me know immediately**.**

Conventions

We will use Times New Roman font for R commands and outputs, with any text in Arial. Note also that you have both an R-console and an R-GUI. Most of the time we'll be using the R-console, but if you expand the drop down menus in the R-GUI you will see that there are some short cuts. One I use a lot is >Misc>remove all objects. Another switches between windows (graphics and console).

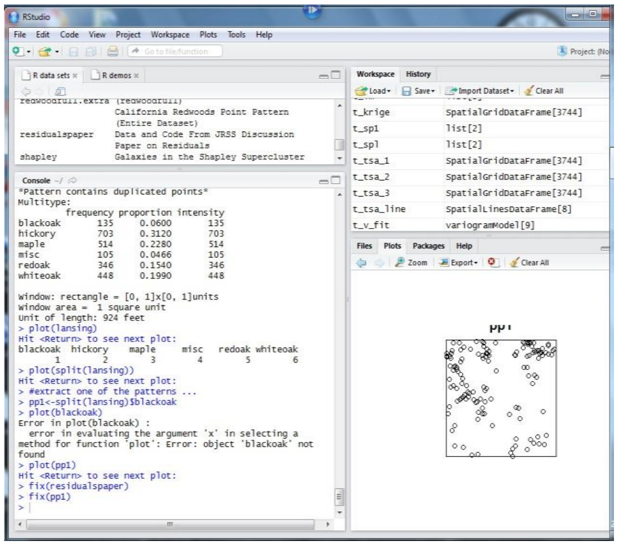
When working through this and the other lessons, it is useful to keep open a window with the lesson and another with the R console open, which enables you to cut from the former and paste into the latter and so saves on typing. If you do this, don't do it mechanically, have a close look at the materials being cut and paste. In fact, the more I think about it, the more I think you should type in all the commands in full.

Using R-Studio

At this stage it is by no means essential, and at times it can complicate matters, but if you intend to write your own R code I'd recommend installing the R-Studio graphical user interface available as a download from:

www.rstudio.org

Once installed you access R via the R Studio icon and, as the screen shot below, shows it provides four adjustable windows. Top left is a source code editor and top right your workspace/history. Bottom left is the R console and bottom right a list of files, plots and so on:


RStudio
File Edit Code View Project Workspace Plots Tools Help
Go to file/function
Project: (No
R data sets
R demos
Workspace History
Load Save Import Dataset Clear All
Console ~/
Files Plots Packages Help
Zoom Export Clear All
pp1

```
redwoodfull.extra (redwoodfull)
                    California Redwoods Point Pattern
                    (Entire Dataset)
residualspaper      Data and Code From JRSS Discussion
                    Paper on Residuals
shapley             Galaxies in the Shapley Supercluster
```

```
*Pattern contains duplicated points*
Multitype:
         frequency proportion intensity
blackoak       135     0.0600       135
hickory        703     0.3120       703
maple          514     0.2280       514
misc           105     0.0466       105
redoak         346     0.1540       346
whiteoak       448     0.1990       448

Window: rectangle = [0, 1]x[0, 1]units
Window area =  1 square unit
Unit of length: 924 feet
> plot(lansing)
Hit <Return> to see next plot:
blackoak  hickory    maple     misc   redoak whiteoak
       1        2        3        4        5        6
> plot(split(lansing))
Hit <Return> to see next plot:
> #extract one of the patterns ...
> pp1<-split(lansing)$blackoak
> plot(blackoak)
Error in plot(blackoak) :
  error in evaluating the argument 'x' in selecting a
method for function 'plot': Error: object 'blackoak' not
found
> plot(pp1)
Hit <Return> to see next plot:
> fix(residualspaper)
> fix(pp1)
>
```

```
t_krige      SpatialGridDataFrame[3744]
t_sp1        list[2]
t_spl        list[2]
t_tsa_1      SpatialGridDataFrame[3744]
t_tsa_2      SpatialGridDataFrame[3744]
t_tsa_3      SpatialGridDataFrame[3744]
t_tsa_line   SpatialLinesDataFrame[8]
t_v_fit      variogramModel[9]
```

1.2 Basic R concepts

Task 1.2 : A quick R workout

If you are familiar with R there is no need to do this, just skip to Section 1.3

As you will see, R can be used as a simple calculator, but it is also a programming language. One of its strengths is that even though it can be used in these ways, it doesn't have to be, and the amount of programming required can be as little or as much as you like. For most of the time we will be using it as a high level command-line driven system to perform quite complex tasks without ever having to know the detail of the many lines of code that are actually being implemented. In this respect it is similar to early, command driven GIS such as the first releases of *Arc/INFO™*. This short introduction is intended to cover the major feature of the environment. If it wets your appetite, as I am sure it will, then *statistics.com* has a number of other courses that use it, including one from Joris Meys, co-author of *R for Dummies* and one from Chris Brunsdon on *Mapping in R*.

If you have a copy of Brunsdon and Comber (2015) this material is covered in rather more detail in Section 2.1-2.3 pages 10- 35

Using R as a calculator

At the ">" prompt type 1+1

```
> 1+1
```

And the answer is returned directly. Now try a few more calculations using the usual symbols (-, +, /, ^, *) noting that brackets can be used to ensure that the order of calculation is as intended. For example try >1+2 * 3 as opposed to >(1+2)*3

For reasons that will become obvious, rather more interesting is sqrt(100), which will return the square root of the argument in the parentheses, factorial(4) which returns the value of 4! and abs(-10) which returns the absolute value (10).

Creating objects and storing a result

Assignment uses the standard *becomes* operator which in R is <- ('less than' followed by a hyphen):

```
>a <-  2.3456
```

This creates an *object* called ‘a’ and assigns the value 2.3456 to it. To view the contents of an object, simply type its name:

```
>a            # display objects content on screen
```

Note how comments can be inserted into the workflow using #

An object can be text:

```
>My_name  <-  “David Unwin”
```

Note that all text in R, including these commands, is case sensitive and that this frequently matters greatly!

Storing sets of numbers

```
>set1 <- c(1, 2, 3, 4, 5, 6)
```

In this set1 becomes a vector of length 6 with the first six integers and c is a reference to an object called c which is actually a method that ‘concatenates’ these numbers into the object. To save typing in long lists of numbers various shortcuts are available, for example:

>Set2 <- c(1,2,3,4,6,7,8,12,13,14) can be shortened to set2 <- c(1:4,6:8, 12:14)

Now try evaluating and printing the object set1 multiplied by 2, which should return 2, 4, 6, 8, 10,12. Doing this illustrates a very powerful feature of R, that it is a *vectorized* language. The command operates on all the elements of the vector, something that in many programming languages would have to be accomplished using an explicit set of looping instructions.

Listing all objects in your current workspace

Use ls()

Removing objects

Use rm(list of objects separated by commas) or, to remove all, rm(list=ls()) or use the R-GUI drop down menu item that does the same thing.

In the second example note how the objects are nested. The object ls() lists all objects you have created in the workspace. In turn its results are assigned to another object called list and it is this that is removed using the object *rm*. Do this step by step starting with >ls() and you will see how it works. The standard R environment has lots of objects for statistical analysis and in many cases a simple guess at the required name will turn out to be correct, but there is also comprehensive help about specifics available using the ? query or help(“object-name”), for example:

```
>?median
```

```
>help ("median")
```

Note that the later opens on-line help (you need to have an internet connection available) in a separate window.

I hope that by now you can see the beauty and above all else the *consistency* of this environment. *Everything* is done by objects with optional lists of arguments passed to them via what is included in the parentheses, e.g.

```
> set1 <- c(1,2,3)
> sum(set1)  returns the answer 6
```

Programming in R thus consists of typing the names of objects followed by any required parameters in the defined order.

Quitting R

Use quit() which gives an option for you to save your workspace in much the same way as using, say, *Excel*™ does

Repeating commands

To save typing note that the "up" and "down" arrows to the right of a standard keyboard will recall previous commands and scroll through them.   You can easily scroll along each command to edit it. This feature of the environment is very useful.

That's really about all you need to know for now!

1.3 Types of spatial data

For the R environment to be useful for spatial analysis we need two related sorts of object:

a) Objects for getting spatial data into the environment in ways that make sense from the perspective of a spatial analysis; and
b) Objects that will perform tasks that we might want, such as producing maps and calculating summary statistics.

Fortunately, both exist because over the past few years a small group of spatial analysts have taken the time and trouble to implement methods of spatial analysis, grouping them into a number of packages, each dedicated to a particular data type or set of analytical methods.   In R terminology a *package* is a group of related classes/objects and often including sample data as well. Some of the more recently contributed packages use a common framework afforded by the sp package that provides classes for handling point, line, polygon and grid spatial data, but many of them do not, and major 'downside' to R is the almost bewildering variety and range of contributed packages (the last time I looked they listed over 8,000). Roger Bivand (Bergen, Norway) who maintains R-spatial refers to this as a 'jungle'. This complicates resource discovery and has also led to some duplication of functionality. It also means that fairly frequently there are several ways to achieve the same results using different packages.   Throughout this course I will use the ones with which I am familiar, but if you see alternatives or more efficient ways of doing things then do let me and the others on the course know by way of the discussion forum.

Task 1.3: Spatial data

Now read Section 1.2 pages 5 -18 of the course text O'Sullivan and Unwin (2010) *Geographic Information Analysis* (NY: Wiley, Second Edition). The First Edition has similar material, pages 4-12 but omits some additional materials on problems with this simple geometric view.

It is common nowadays to recognize four types, or classes, of geographic data according to their geometric dimension of length (L):

1. *Points* (L^0), often called *event data* by statisticians;
2. *Lines* (L^1) (with one exception not considered further in this course, see page xiii of the Preface to the Second Edition for our reasoning on this);
3. *Areas* (L^2) or in GIS *polygons* and in statistics *lattices*; and
4. *Fields* (L^3), sometimes *surfaces* and in statistics *geostatistical data*.

Task 1.3 above should have convinced you that even this simple view of geography can be disputed and it is wise to be aware of the limitations of these categories.   Whether or not any of the qualifications mentioned in the text are important is really a matter for the specifics of any particular instance and is something you might like to think about.

There are R packages for the analysis and display of each of these data types. The main ones you are likely to want to use for the analysis of spatial data are:

- Point data: spatial, *spatstat, splancs, VR:spatial spatialEEpi*
- Lattice data: *spdep, DCluster, spgwr, ade4, GISTools, GWModel*
- Geostatistical data: *gstat, geR, geRglm, fields, spBayes, RandomFields, VR:Spatial*, *sgeostat, varchag*
- Gridded raster data: *raster*
- General utilities for spatial data: *sp, RgoogleMaps, rgl, rgeos, rgdal, RColorBrewer, OpenStreetMap, maptools*

There are others! We could of course install them all at the outset and just assume that what we need is available, but I think it best to install as we need them. This may be a little inefficient, but at least it gives some insight into their various dependencies.   If you so wish, you can install a specific task-view of the R environment using >ctv followed by install.views("Spatial").

In this course, Lesson 2 will address methods for analyzing point objects, Lesson 3 lattices, and Lesson 4 geostatistical data, in each case using an appropriate R package.

1.4 Spatial data and the R environment: point (aka: event)data

In the RGUI (not the console) there is a tab packages. Click on this and in the drop down menu select install packages. When you do this R will ask you to select a CRAN Mirror site from which to download. You have the option to download from about 20 sites that offer secure transfer (HTTPS) and at the bottom a much longer list of HTTP sites accessed by clicking on *HTTP mirrors*. As a UK resident I tend to use the one at Bristol University, but you should use one close to your home or one that is likely not to be very busy. Doing this brings up a long list of available packages additional to those loaded as part of the base R system. For point data there are three main packages called spatial, spatstat**,** and splancs.

During the install you will be asked to create a *personal library*: do this. If you do not you may well find that packages will not be downloaded because you do not have the correct administration privileges.

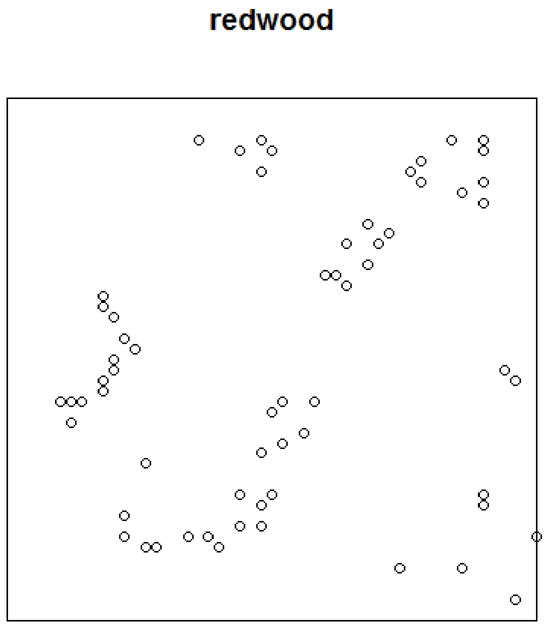
In this first lesson we will use the package spatstat developed by Professor Adrian Baddeley in the University of Western Australia (Perth) and Rolf Turner who is in the University of Auckland (NZ), but the lesson could have equally well been developed using splancs developed by Professor Peter Diggle at Lancaster University, UK.

Go to your chosen CRAN mirror site and install spatstat into your personal library. When it is installed >??spatstat will lead via WWW to a full description of the package with vignettes of its use, code demonstrations and so on. The file spatial_package gives a clear account of what it can do. The package has bundled in it a series of sample data sets, of which we will use the locations of 62 redwood tree seedlings in a small, 23m x 23m square area. These data have been used many times, most notably by Peter Diggle (1983) himself.

```
> library(spatstat)
> data(redwood)
> summary(redwood)

Planar point pattern: 62 points
Average intensity 62 points per square unit

Coordinates are given to 3 decimal places
i.e. rounded to the nearest multiple of 0.001 units

Window: rectangle = [0, 1] x [-1, 0] units
Window area =  1 square unit
```

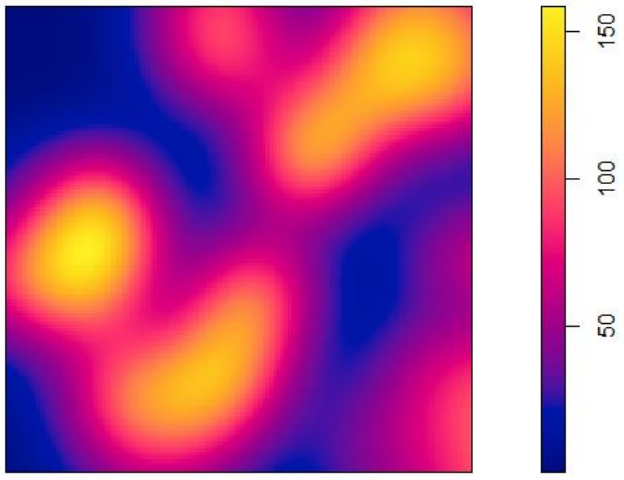
This simply confirms that you have these data. Now we plot them in a separate window as a simple pin/dot map:

```
> plot(redwood)
```



It's as easy as that! Note that in this case the system knows that the object redwood is of the object type called a *planar point pattern* (ppp) so the generic command plot selects and uses the appropriate method for this type of object. Had redwood been some other type of object, a different method would have been selected and applied. In the RGUI you can save in any one of several picture formats using >FILE>SAVE AS and following the usual Windows dialogue.   I usually set up a folder called something like results_data and save to it as I go along. It is rather nice to realize that you can visualize this pattern easily using a *kernel density* estimate (KDE) by creating a pixel image object as

```
>z<-density(redwood, 0.1)
>plot(z)
```



What is happening here is that we are using a method called density to compute a fixed-bandwidth kernel estimate of the intensity function of the point process underlying a planar point pattern and store it in a pixel image object we called z:

```
> summary(z)
real-valued pixel image
128 x 128 pixel array (ny, nx)
enclosing rectangle: [0, 1] x [-1, 0] units
dimensions of each pixel: 0.00781 x 0.0078125 units
Image is defined on the full rectangular grid
Frame area = 1 square units
Pixel values
	range = [0.08964096, 158.2837]
	integral = 64.66244
	mean = 64.66244
```

Note that in the entire R environment, there are lots of methods that can be called using density.

The default density that we used computes the (strictly speaking 'convolution of') an isotropic Gaussian kernel of standard deviation sigma set here at 0.1 distance units (or one tenth of the distance along both *X*- and *Y*- axes) and with each point having unit weight. The same generic function can have numerous arguments that control how the KDE estimates are derived for other types of object, but in spatstat you are stuck with just a Gaussian kernel.

If you have got so far it's nice to add the points to the KDE:

```
>plot(redwood, add=TRUE)
```

Task 1.4: Maps and mapping (10 points)

I'd like to review and assess things so far, so this Task will be our first graded assignment, worth 10 points (from 25 overall for this lesson). If you replicate the above sequence you will have generated at least two maps of the redwood point data, the first a pin/dot map, the second a continuous surface of estimates of the intensity of the spatial process.

Maps are a major tool for the analysis of spatial data, but they are often misunderstood and can often be very deceptive. When David O'Sullivan and I wrote the first edition of *Geographic Information Analysis* round about 2000/2001, we did not feel that we had to cover maps and mapping in any detail, regarding it as something that could be 'taken as read'. Experience since then, some of it with *statistics.com*, has shown that this assumption was wrong and that it pays to have a close look at the entire mapping process. For this reason, in the Second Edition of the text, we have added a wholly new Chapter 3 with the title *Fundamentals – Mapping it Out*. Particularly if you haven't had much contact with cartography ('the art and science of mapping') it will be useful to take a break from the computer as follows:

a) Read Sections 3.1-3.5 pages 55-66 of O'Sullivan and Unwin (2010) *Geographic Information Analysis*, that deal with general issues;

b) Read Section 3.6, pages 66-72 'Mapping and Exploring Points' that outlines the visualization method of kernel density estimation. The help in spatstat will also be useful. The First Edition of the text has some of these materials scattered throughout, but it will be simpler if you let me know and I will supply a copy of the entire newly-drafted chapter;

c) Next, create a WORD (or similar) file and use INSERT>PICTURE to insert a copy of your 'dot' map of the redwood data. Add a comment in which you describe the pattern so revealed. Would you say it is 'random', 'clustered' or 'more regular than random / dispersed'?

d) Finally, experiment with density using differing bandwidths and insert what you think is the map that best visualizes the pattern, adding a commentary on the reasoning behind your choice. Why not post your view on this on the course Bulletin Board?

Mapping point/event data is covered in Comber and Brunsdon, Sections 3.4.4 pages 71078. KDE is covered, Sections 6.2.-6.4. Bothe sections use alternative package, Brunsdon's own *GISTools*

Note that this KDE function is often misunderstood. It is not a spatial smoothing of any data or weights attached to the point pattern as would be the case in the interpolation of a continuous field from sample geostatistical point data. (To do a spatial interpolation of values that were observed at the points of a point pattern in spatstat one can use smooth). Nor is it a probability density. It is an *estimate* of the *intensity function* of the point process that generated the point pattern data. *Intensity* is the expected number of points per unit area and its units are 'points per unit area'. The integral of the intensity function over a defined spatial region gives the total expected number of points falling in this region. Inspecting an estimate of the intensity function is usually the first step in exploring a spatial point pattern dataset. For more explanation, see the *Workshop* notes in Baddeley (2008) or Diggle (2003).

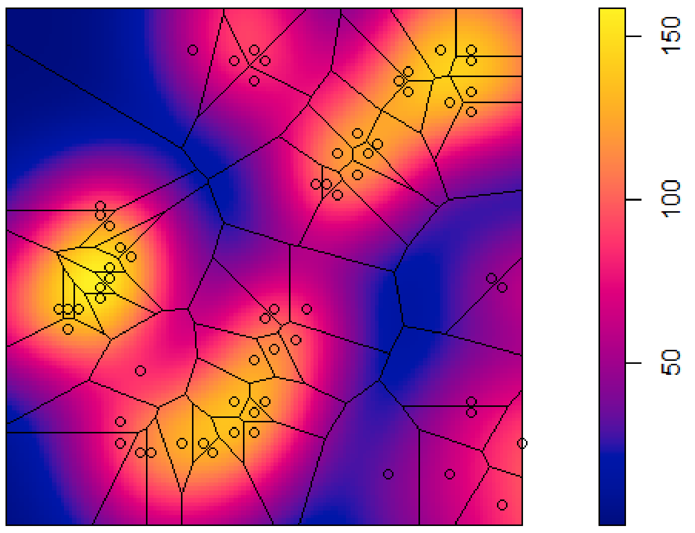
Task 1.5: What can spatstat do?

It is useful at this point to review what is available in spatstat. To do this call for the on-line help to access the package description and read it, using:

```
> help("spatstat") or
>??spatstat
```

Try invoking the series of simple operations listed below to rotate and shift, compute two tessellations of the point pattern, and conduct some simple global tests using as standard the idea of complete spatial randomness (CSR) (See O'Sullivan and Unwin , 2010, pages 130-152).

```
> t<-rotate(redwood)
> plot (t)
> s<-shift(redwood)
> plot (s)
#now generate a display of KDE, point pattern and proximity polygons
>plot(z)
>d<--dirichlet(redwood)
> plot (d, add=T) # NB upper case for TRUE shortened to T
>#add the points …..
 > plot(redwood, add=T)
```



```
> d<-delaunay(redwood) # NB the spelling used
> plot (d)
> nna<-clarkevans(redwood) # this is the standard ‘nearest neighbor statistic for these data with an edge
correction due to Donnelly
> nna
    naive       Donnelly       cdf
0.6186502    0.5849906    0.5835880
```

So these data are ‘clustered’? But isn’t this all just a bit too easy?

Getting point data into *spatstat*

Yes, it is easy because the data already are in the required format for the package as an *object* of *class* ppp. I suspect that almost all the problems you will encounter in the future when using R with spatial data will be to do with getting the required data into the environment in ways that the particular package you are using can understand. The major issue in developing R-spatial lies in the fact that, by and large, statisticians are happy to consider their data organized as simple tables in which rows represent the cases and columns the variables, much as, for example, in a spreadsheet workspace, whereas, for various equally perfectly sensible reasons, spatial data analysts require more complex formats. Spatstat was written before any attempts on standardization (which resulted in the sp package, see later) and so uses its own object class called ppp. The immediate problem resolves itself into getting any external point data into ppp format.

Getting a text file into *spatstat*

This is accomplished using the basic R function read.table together with a conversion object called ppp to create a data object of class ppp that spatstat can understand. We will illustrate the approach using a text file snow_for_R that should be downloaded from *statistics.com*'s website for this lesson (it is linked from the main lesson page) and stored in a convenient place on your machine:

```
> library(spatstat)
> # loading data from an external file. Note use of full path and double separators as in \\
>  snow<- read.table("C:\\Users\\David\\Downloads\\SNOW_for_R.txt", header=TRUE)
```

(A version of these same very famous and much analysed data can be found at: http://cran.r-project.org/web/packages/HistData/HistData.pdf)

Note use of full path appropriate to the way I have set this up on my machine and *double separators*, as in \\. You will need to change whatever is within the quotation marks (" … ") to give the path for your machine. Experience suggests that finding the right file and getting the spelling and punctuation right can be a big stumbling block, so take care! There is an element of personal choice here. You could put these data into your current working directory using >setwd ("directory") but for reasons best known to myself I prefer to copy the full file path using the standard right click for properties in Windows and then edit the \\ in after I have pasted it into the R command. Your choice!

The original text file *snow_for_R* starts :

```
   X            Y
13.58801      11.0956
9.878124      12.55918
14.65398      10.18044
15.22057      9.993003
13.16265      12.96319
```

etc for 578 rows

and the assignment using read.table creates another object that we call just snow. We need first to attach it so that we can read individual (*x* , *y*)_ coordinates in the columns **X** and **Y** as:

```
> attach(snow)
> summary (X)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  8.281  11.640  13.210  13.030  14.520  17.940
> summary (Y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6.09   10.68   11.52   11.70   12.76   16.97
> nrow(snow)
[1] 578
```

The key to using spatstat for point pattern analysis lies in the object class ppp and associated method ppp that creates an object of this class. It can have many arguments, but the simplest are just the vectors of **X** and **Y** co-ordinates and the co-ordinates of the required bounding box.

Supposing that we aren't really sure about these, we could use the summary data from above to create the xlim and ylim objects and then pass these to ppp:

```
> #create PPP object for use in spatstat
> xlim <- c(8.0, 18.0) # a little bigger for the bounding box
> ylim <- c(6.0, 17.0) #also a little bigger
> snow_2 <- ppp(X, Y, xlim, ylim)
Warning message:
In ppp(X, Y, xlim, ylim): data contain duplicated points
```

Finally we can draw a pin/dot map and analyze these data as we see fit:

```
> plot(snow_2)
```

and over to you …

Task 1.6: Snow's data (5 points)

Acquire the snow_for_R text file, and convert it into a ppp object as above and then produce a dot/pin map of the distribution. Add your resulting map to your WORD file and again comment on the distribution. Using density with a sensible bandwidth also helps visualize the pattern. You can of course superimpose the points using add=TRUE).

Perhaps you'd like to suggest on the discussion which bandwidth gives the 'best' picture?

The name has perhaps given the game away, but this is probably the most famous point data set ever to be analyzed. It consists of the locations of 578 deaths from cholera recorded by Dr. John Snow in the Soho area of London during an outbreak of the disease in 1854. Snow mapped these data as a dot/pin map and was able to show that they clustered around a single water pump (no piped water in those days!) in Broad (now Broadwick) Street. Acting on his advice, the authorities removed the handle from the pump and the epidemic ended soon after, although it may well have already been past its peak. The events are celebrated by a facsimile of the pump in Broadwick Street and in the naming of a nearby pub as the *John Snow*. All epidemiologists and spatial analysts should at some time make a pilgrimage to the street and have a drink in the pub that now bears his name.

Snow's work is widely regarded as the birth of scientific epidemiology, and his demonstration that the vector for cholera was water-borne led to massive investment in UK during the second half of the nineteenth century to provide safe public water supplies. Of course, the story isn't as simple as it is sometimes suggested. For a recent scientific account, see Brody, H. *et al*., (2000) Map-making and myth-making in Broad Street: the London cholera epidemic 1854. *The Lancet*, 356 (9223):64-68. For a popular account, I highly recommend Steven Johnson's gripping account of the incident in his book *The Ghost Map*.

Given that we took so many pains to create a file of data that spatstat can understand it makes sense to save it into the current working directory using *set working directory*, setwd, and save, for example:

```
>setwd("C:\\Documents and Settings\\David Un win\\Desktop\\R-results")
>save(snow_2, file="snow_in_ppp")
```

This will save the file (now in binary, but this can be changed to ASCII, for example, to enable access other software) in the defined working directory.

A comment on formats

The object class ppp doesn't really add much 'geography' to the list of co-ordinates making up the point pattern, since all it does is define the bounding box in the same co-ordinate system used for the data. Typically, this might be numbers scaled to the interval (0,1) along both axes, or some simple equivalent sequence such as (0, 100). For much analysis this is likely to be perfectly adequate, enabling almost all the usual point pattern statistics to the calculated, but think for a moment about its more general use. If we really were interested in the underlying geography (for example to identify causes of inhomogeneity in the process) we'd almost certainly want to relate these data to some other real world geography such as a topographic map or even aerial photography from *Google Earth*™. Clearly, we could do this by renumbering the co-ordinates into some real world system and possibly also projecting the pattern onto some standard map projection, but this would be messy. The alternative is to embed the required information in the format used to store the spatial data at the outset. The sp package developed by Roger Bivand and Edzer Pebesma provides tools with which to do this. First, go to packages and install sp from your selected CRAN mirror and install it in the same way that we installed spatstat. A description of what sp includes is given by:

```
> library (sp)
> getClass ("Spatial")

Class "Spatial" [package "sp"]

Slots:

Name:       bbox proj4string
Class:    matrix        CRS

Known Subclasses:
Class "SpatialPoints", directly
Class "SpatialLines", directly
Class "SpatialPolygons", directly
Class "SpatialPointsDataFrame", by class "SpatialPoints", distance 2
Class "SpatialPixels", by class "SpatialPoints", distance 2
Class "SpatialLinesDataFrame", by class "SpatialLines", distance 2
Class "SpatialGrid", by class "SpatialPoints", distance 3
Class "SpatialPixelsDataFrame", by class "SpatialPoints", distance 3
Class "SpatialGridDataFrame", by class "SpatialPoints", distance 4
Class "SpatialPolygonsDataFrame", by class "SpatialPolygons", distance 2
```

Note that the class defines both a bounding box (bbox) and the coordinate reference system (CRS) in which this and the (x, y) co-ordinates are defined. The coordinate reference system is defined by a string that can be NA (= not available) or make use of the *PROJ.4 Cartographic Projections Library* (see *http://proj.maptools.org* for more details). As we are about to see, sp comes in very handy for a quick look at some lattice data.

1.5 Spatial data and the R environment: lattice (aka polygon) data

Lattice data refer to lattices of polygonal area objects that usually are planar enforced, that is, they tessellate the plane without overlaps or gaps, as for example in a map showing the borders of the States of the USA or of the Counties within a State. In order to analyze and map them, we not only have data on attributes of these areas, but also on their geometry. A simple R plot of the boundaries of the 100 counties of North Carolina is shown below:

These outlines were recorded as geographical co-ordinates (latitude, longitude) and have been ‘projected’ onto a flat sheet of paper so that the result is inevitably a distortion of what we might think of as the ‘true’ shape of the state. Right now for our purposes this doesn’t matter, but the moment we tried to associate the data with other geographical information (for example if we wanted to ‘mash’ them onto Google Earth™) we would have to ensure that both they and any Google image are recoded on the same coordinate reference system.

It will be apparent that assembling any attribute data for these counties is easy, all we need is a standard data table consisting of a single row for each county with two columns, one for an identifier for that county and one for the value of any variable attached to it. But what about the geometry we need to create a map such as that above? First, we’ll need a lot of numbers to define each and every line segment that give the county borders, and, second, almost certainly we’d capture these data using either off screen digitizing from an image or from a semi-automatic line digitizer. In the early days of computerized cartography much was made of the labor needed to do this and the difficulty of ensuring that the data obtained were free from errors, but nowadays almost all such geometric data can be obtained from one or other of many libraries that hold them ready to be used.

The format that is nowadays almost the *de facto* standard for exchanging geometry data of this sort is the *shapefile* (.shp) that was originally specified by ESRI™ for its *ArcView*™ GIS and mapping system. In fact, the shapefile formats (there is more than one according to the type of geographical entity) uses several separate tables including an index file for the geometry (.shx), the geometry itself (.shp), and a file for the attributes in a very old format used by the original *dBASE* data management system (.dbf). Sometimes there is also a .prg projection file. You might like to think about how such data could be forced into the simple tables provided in R by the data.frame concept.

(There are lots of ways in R to access the data in a shapefile and to write data back into the same one. I have always used the shapefiles package and related packages as in the Lessons. Googling will quickly show lots of materials on the functions provided, for example, see:

- http://cran.r-project.org/web/packages/spatstat/vignettes/shapefiles.pdf
- http://cran.r-project.org/web/packages/shapefiles/shapefiles.pdf
- http://www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html

I find the Lancaster tutorial very useful)

Go to the GUI and click on Install Packages and install maps, maptools and if not already in your library sp as well:

```
> library(maps)
> library(maptools)
>#and if not already loaded …
> library(sp)
```

In doing this you might as well be on the safe side and install the gpclib using >gpclibPermit when this is requested. Next we use the readShapePoly with appropriate arguments, assigning the data into an object we call sids_2 :

```
>sids_2 <- readShapePoly("C:\\Users\\David\\Downloads\\sids2\\sids2", IDvar="FIPSNO",
proj4string=CRS(as.character(NA)))
```

There is a lot to take on here that we will deal with as and when we need to, but for now just concentrate on getting these data into your machine. Note that in the above you will again need to specify the absolute path to where the sids2 shapefile is stored on your machine. Having got sids_2 from a shapefile into something R can handle, these data can be mapped using both the traditional graphics system and extensions to it implemented in the sp package.

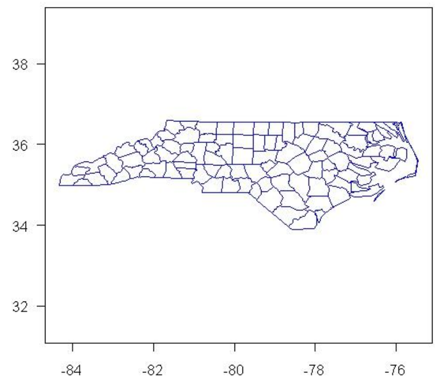
In the traditional system:

```
> plot(sids_2)
```

… will plot the polygon boundaries. The result can be 'improved' in various ways by additional arguments to control the style and layout, for example:

```
>plot(sids_2, border="red", axes=TRUE)
```

This produces a map showing the county 'lines' in red with the axes labeled:



Task 1.7: Maps for lattice data

Now take time off from your machine and read Section 3.7 *Mapping and Exploring Areas*, pages 72-79 of O'Sullivan and Unwin (2010). The issues raised in this section are elaborated and illustrated in a 2001 web essay by Jason Dykes and I with the title *Maps of the Census: a rough guide* at *http://www.agocg.ac.uk/reports/visual/casestud/dykes/conten_1.htm* (checked 25/11/15)

Mapping lattice data using the choropleth method in GISTools is covered in Comber and Brunsdon, Sections 3.4.1-3.4.3 pages 65-71

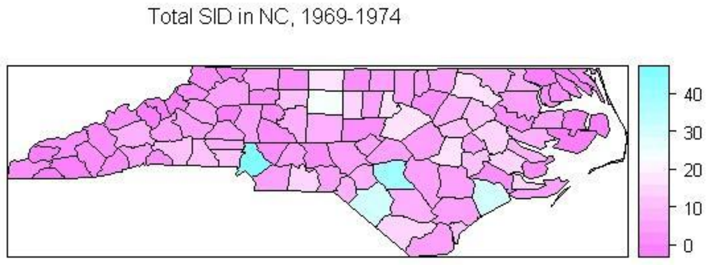
Before we start drawing any maps, it is useful to list the attribute data available in the sids_2 spatial data frame:

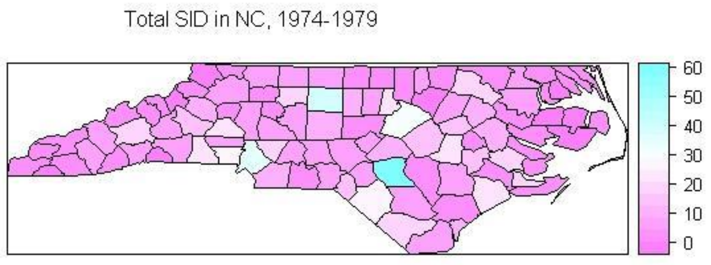
```
> sids_2  # prints out the entire contents of the object sids_2. Note how both the attributes and the geometry are stored.
> as(sids_2,"data.frame")  # this forces the object to belong to the specified class and will echo just the attribute data to the console
```

Note that we have information for each county on its AREA, PERIMETER, COUNTY (CNTY ID), FIPS_NO (a national numbering system), the identifier used by Noel Cressie in his book *Statistics for Spatial Data* (CRESS_ID), Total births in 1969 -1974 (BIR74), Sudden Infant Deaths 1969 -1974(SID74), Non-White Births 1969 -1974 (NWBIRT74), Total births in 1974 -1979 (BIR79), SIDs in 1974 -1979 (SID79) and Non-White Births 1974 -1979 (NWBIR79).

Once we have our shapefile as a data.table in R very basic choropleth maps can easily be produced using the spplot method available in the sp package. The two maps in the panel below showing the Sudden Infant Death Syndrome (SIDS) totals for in each county of North Carolina from 1969 to 1974 and from 1974 to 1979 were produced using the sids_2 object as:

```
> library(sp)
> spplot(sids_2,"SID74")
> spplot(sids_2,”SID79”)
```





Both maps were titled using Microsoft *Paint*™ and they use the default number of classes (20) and color ramp in spplot. The optional arguments col.regions = and at = allow you to change these, and the classInt package has facilities for choosing class intervals.

For advice based on many years of cartographic experience, Cindy Brewer and Mark Harrower's website at *http://colorbrewer2.org* (checked 20/10./14) is worth a visit and in the R environment the package RColorBRewer provides the palettes described by Brewer *et al.* (2003) for continuous, diverging, and categorical attributes. Having got the basics, please feel free to experiment to produce visually more exciting maps! If you really need 'publication quality' choropleths then *GISTools* has all the methods you need.

Task 1.8: Your own choropleths (5 points)

1. There are some other variables in the sids_2 data giving the totals numbers born (BIR74, BIR79) and non-white births (NWBIR74, NWBIR79). Use the approach outlined above to produce a map showing one of these variables and comment on the distribution it shows;
2. The maps reproduced above and that you have created under (1) above has one massive flaw. Tell me what it is. You might find that consulting an atlas map of the State helps the interpretation;
3. Finally, what would you need to do to fix this and create more informative maps?

1.6 Spatial data and the R environment: geostatistical (aka field/surface) data

Another type of spatial data to consider is the continuous, self-defining, scalar *field* or *surface* in which everywhere has a value and where (unless the field is generated by a mathematical formula of the form $z = f(x, y)$) our data are a sample of the field's height/value at some known location. Usually these heights are symbolized as a *z*-value (not to be confused with a standard *z-score*). Statisticians call such data *geostatistical*. Reconstruction of the shape of the field from these sample data points is called *interpolation* and the art and science of doing this makes use of a series of statistical methods that are collectively called *geostatistics*. Display of reconstructed fields typically uses the simple (?) *contour-type* of map in which lines of equal value of the height variable are threaded through the data and projected onto the (x, y) plane at $z = 0$. In the literature these contour-like lines go by various names that have the prefix *iso* (=same), such as iso-therm for a field of temperature values, iso-hyet (rainfall), iso-bar(atmospheric pressure) and so on.

Task 1.9: Mapping and exploring fields

Now read Section 3.8 *Mapping and Exploring Fields*, pages 80-84 of O'Sullivan and Unwin (2010). If you routinely use geostatistical data, my former student Jo Wood's *Landserf* package (page 84) is well worth trying out.

You will need the maptools library loaded to be able to access some geostatistical data for trace elements in the soils of the Meuse river valley, the *lattice* graphics package (which comes as part of the base R system) to display them, and the sp package for some of the other manipulations. The data are already available in sp.

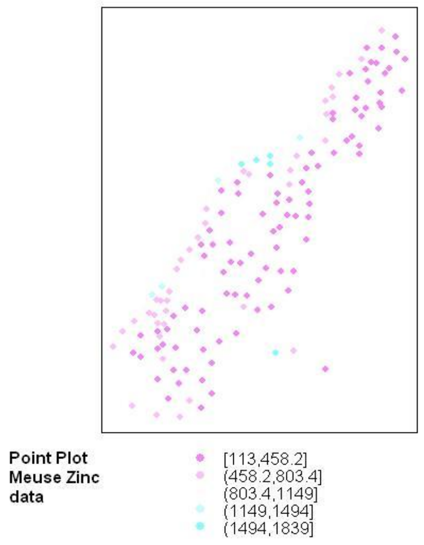
```
>library(maptools)
> library(sp)
> library(lattice)
> data(meuse)
> meuse
       x      y cadmium copper lead zinc  elev       dist   om ffreq soil
1   181072 333611    11.7     85  299 1022  7.909 0.00135803 13.6     1    1
2   181025 333558     8.6     81  277 1141  6.983 0.01222430 14.0     1    1
3   181165 333537     6.5     68  199  640  7.800 0.10302900 13.0     1    1
```

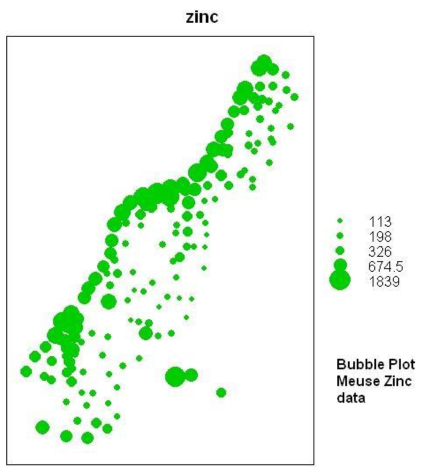
etc,

The Meuse data has164 data points with nine possible height/value variables including factors coded as integers and a character string. Two “honest” ways of displaying these data in lattice graphics are *color coded maps of the points* in which the color intensity is set on a color ramp (as in choropleth mapping) and a *bubble plot* in which proportional circles are used to give what in the course text, pages 71-72 we call a *located proportionate symbol map*. We also need to identify the spatial cooridnates as the “x” and “y” variables in the data frame:

```
> coordinates(meuse)<-c("x", "y")
> spplot(meuse,"zinc")
> bubble(meuse, "zinc")
```

This produces the two maps shown below, again using the default colors and class intervals.





For reasons I do not understand Comber and Brunsdon refer to this type of display as a ‘choropleth point map’. They show how to map them on pages 75-77

There is something fairly important about the maps, and that is that they are in some sense “honest”, with a color coded symbol or proportionate symbol located at the each of the (*x, y*) co-ordinates of the sample points. No attempt has been made to extend these data by interpolation out into the spaces between the data points and no contours have been threaded through them. Threading contours (interpolation) is less honest because there are many ways by which we could it, each producing (hopefully, but not always) small differences in the field so created.   To my mind it is wise to be aware of this and to leave interpolation for another day (in fact Lesson 4).

Getting a text file into R as a SpatialPointsDataFrame object

Most of the time you won’t be interested in the demonstration Meuse data that comes with R, and we will need to be able to import our own data. One option is to use ESRI *Shapefiles* prepared in *ArcGIS*™ (or for that matter in *GeoDa*™ which will output the same thing) using the same approach as that used to import a polygon data file in Section 1.5. Another is to import directly from a text file containing the required data. Either way, the objective is to create a SpatialPointsDataFrame object in a format that R can understand.

The text file topo_data.txt has data for 50 spot heights of surface relief in a small 300 x 300 foot area and is structured:

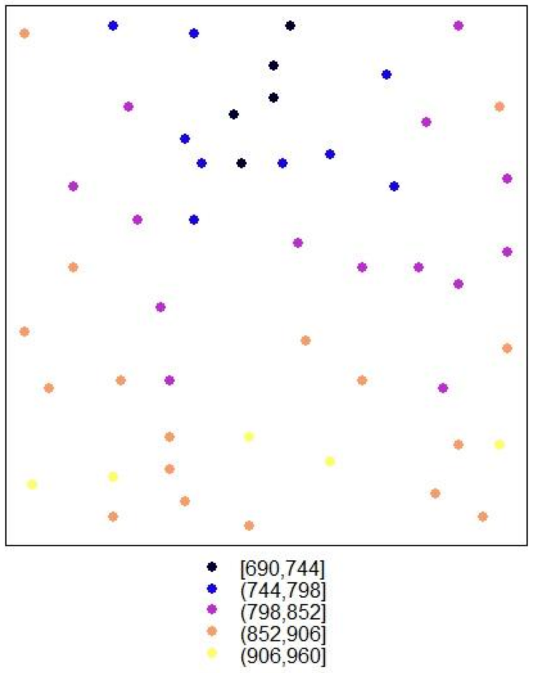
X	Y	Z
0.3	6.1	870.
1.4	6.2	793.
2.4	6.1	755.
3.6	6.2	690.
5.7	6.2	800.

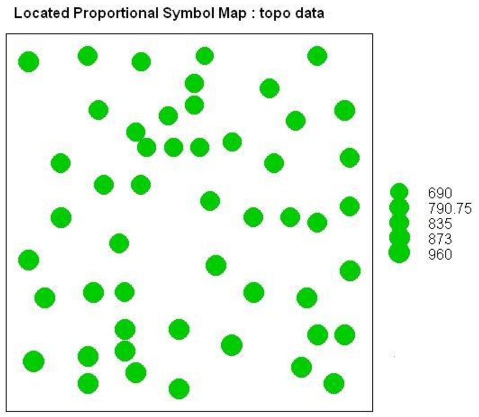
```
etc to
5.7     3.1     830.
```

Note that we have no row identifiers and just one height or *z*-value. The coordinates are in some local system and are not projected. We can import them using the functions in the sp package as follows:

```
>topo_df <- read.table("C:\\Users\\David\\Downloads\\Topo_data.txt",header=TRUE)
> #check its there
> topo_df
    X   Y   Z
1  0.3 6.1 870
2  1.4 6.2 793
3  2.4 6.1 755

Etc

50 5.7 3.0 830
># extract the x, y values using cbind and put them into topo-mat
> topo_mat <- cbind(topo_df$X,topo_df$Y)
#check its OK
> topo_mat
      [,1] [,2]
 [1,]  0.3  6.1
 [2,]  1.4  6.2
 [3,]  2.4  6.1
etc
 [50,]  5.7  3.0
>#tell it something about the CRS used, even if it isn't …
># NA simply says that we don't know, or aren't concerned to know on that map projection
># these data were recorded but this has to be in the spatial points data frame we create …
> cord_ref <- CRS(as.character(NA))
># create the SDPF object
> topo_spdf <-SpatialPointsDataFrame(topo_mat, topo_df, proj4string = cord_ref)
># display these data ….
> spplot(topo_spdf,"Z")
> bubble(topo_spdf,"Z")
```

The last two commands produce the 'maps' shown below (with titles added using Microsoft *Paint*™). I would be the first to acknowledge that these aren't very good cartographically speaking but at least they are 'honest'. It should be clear that if we want to visualize the surface these data represent we need to display the complete surface using a process of *interpolation* to estimate the surface height over the entire mapped area. Lesson 4 introduces the three most important methods of interpolation.





Task 1.10: Los Angeles Basin atmospheric pollution (5 points)

The text file Simple_LA_OZ.txt has data for atmospheric ozone at 32 monitoring sites over Los Angeles that is structured:

X	Y	Z
0.28122	0.282666	0.15596
0.10093	0.306979	-0.45451
0.15755	0.110818	-1.30413
etc		
0.31455	0.286843	0.62355

These data were downloaded from Dr Luc Anselin's site at *http://geodacenter.asu.edu/sdata*, originally as an ESRI *Shapefile* with complete co-ordinate information and with several variables related to ozone concentration. To create this simple file I have first converted the height variable of interest, which is the average ozone concentration on the eight highest daily values during the observation period, into *z*-scores (note that this will generate negative values of z for values below the mean). I have also scaled both the axes to lie in the same (0, 1) interval. These changes will not affect any of the symbolism used to display these data, but will of course affect the actual values used. Converting sampled continuous surface data into *z*-scores is something I often do, and it is a transformation that can be very useful when comparing maps.

Your final task is simple: import these data into R and produce a located symbol plots (color point and bubble plots) and briefly comment on what this first look at these data suggests. As in previous Tasks, copy these results into your WORD file for submission.

As you do this exercise, I think you will realize the value of being able to associate this map with some geographical context, such as a *Google Earth*™ image or, more simply a coastline. For reference, the bounding box of these data is from latitude 33.6275N to 34.6901N and from longitude -116.234W to -118.535W.

1.7 Spatial grids

The alert reader will have realized that of the family of objects in the sp package we have only really covered SpatialPoints and SpatialPolygons with their data frames SpatialPointsDataFrame and SpatialPolygonsDataFrame. This leaves SpatialLines and the SpatialLinesDataFrame, which we will not consider further, together with SpatialPixels (SpatialPixelsDataFrame) and SpatialGrid (SpatialGridDataFrame) both of which will appear later. Spatial *grids* are regular objects used to represent a continuous layer of information often called a *raster*. Such representation is typical for continuous remotely sensed data, digital elevation models of the earth's surface, and interpolated data from point measurements, but it can also be used to represent categorical variables. At first sight, and for many analyses, such data can be treated very simply using the notion that the grid of square *picture elements* (pixels) is the same as a table in most computing environments and R is no exception. In this simple picture, all that you need to know to register such data onto some real world geography are the grid origin, the number of rows and columns and cell size.

This picture of the grid of equal-sized uniformly square pixels tessellating the plane is one that you need to be a little careful about. First, (see page 189 of O'Sullivan and Unwin, 2010) variations in satellite orbits and aircraft attitude mean that the allegedly uniformly square pixels from remote sensing are unlikely to be exactly so. Second, there is nothing sacrosanct about the square shaped grid cell. Although they nest without gaps or overlaps and can be grouped together into 4's, 16's etc, making analysis possible at differing resolutions, they do not have what is called *uniform adjacency*. It is further from grid cell center to center across a diagonal than it is along the rows and columns. In contrast, uniform hexagons also tessellate the plane and each neighbor center is exactly the same distance away as every other, but these cannot be nested. At least one experimental GIS based on hexagonal rasters has been described. In addition, you might like to think about the possibility of a representation based on uniform triangles and about how such schemes could be implemented in the R environment.

1.8 Conclusion

I would be the first to admit that this is a long, hard,session but hopefully we have all arrived safely at this point. Well done! Life should get easier from now onwards.

1.9 Data files used in this lesson

- Redwood (bundled with spatstat)
- SNOW_for_R.txt
- SIDS.shp (from Geodatacenter @ asu)
- Topo_data.txt
- Simple_LA_OZ.txt (as modified by D. Unwin from Geocenter @ asu)

1.10 References

If you have access to them, any (or all?) of the following will be of interest:

Baddeley, A. (2008) *Analysing spatial point patterns in R. Workshop notes*. CSIRO online technical publication. URL: www.csiro.au/resources/pf16h.html (checked 20/10/14)

Baddeley, A. and Turner, R. (2005a) Spatstat: an R package for analyzing spatial point patterns. *Journal of Statistical Software* 12:6, 1–42. URL: www.jstatsoft.org, ISSN: 1548-7660.

Baddeley, A. and Turner, R. (2005b) Modelling spatial point patterns in R. In: A. Baddeley, P. Gregori, J. Mateu, R. Stoica, and D. Stoyan, editors, *Case Studies in Spatial Point Pattern Modelling*, Lecture Notes in Statistics number 185. Pages 23–74. Springer-Verlag, New York, 2006. ISBN: 0-387-28311-0.

Brewer, C.A., Hatchard, G.W., and M.A. Harrower (2003) Colorbrewer in print: a catalog of color schemes for maps. *Cartography and Geographic Information Science*, 30: 5-31.

Comber, L. and Brunsdon, C. (20-15) *An Introduction to R for Spatial Analysis and Mapping* (London: SAGE)

Diggle, P.J. (1983) *Statistical Analysis of Spatial Point Patterns* (London: Academic Press)

Diggle, P.J. (1985) A kernel method for smoothing point process data. *Applied Statistics* (Journal of the Royal Statistical Society, Series C) 34 (1985) 138–147.