# Cases and Controls, Points and Processes across a Landscape

Uppsala MA Course: Landscape Analysis and GIS

Andrew Bevan
(with past contributions by Enrico Crema & Mark Lake)

R is one of the best and most popular environments for statistical analysis (see http://www.r-project.org/). Its popularity is due partly to the commitment of a core group of developers who are established and well-regarded statisticians, and partly also to the fact that, as a piece of open-source software, it offers a methodologically transparent, applied language for statistical computing that can be easily extended with new functionality, and has a simple command line structure that handles large datasets and runs multiple (batch) processes well.

This tutorial provides a very brief introduction to R, and then switches gear to look at some more advanced point pattern analysis techniques. We use the base R working environment, because there is a virtue in learning how to work in this environment without any additional layers on top of it. That said, additional software layers such as Jamovi (https://www.jamovi.org/, which provides a nice spreadsheet-like interface for R), or RStudio (https://posit.co/products/open-source/rstudio/ which provides a more complicated but also more sophisticated 'integrated development environment' or IDE with features such as 'markdown notebooks') can sometimes be very helpful too. For now, however, we focus on base R.

## 1 The R environment

### 1.1 Starting Up...

1. First make sure that you have downloaded and installed na up to date version of the R statistical package from http://www.r-project.org/. In passing, it is worth mentioning that you will be asked, as a one-off, to choose a 'mirror' site where you make the download from. These mirror sites replicate exactly the same repositories of R code and add-on libraries for reasons of software resilience and speed of access. You can choose anyone you like, but probably best to choose one that is physically close to you.

2. Now please create a folder for this week's work on your computer, for example 'week5'.

3. Now launch R from the Start menu on your machine.

### 1.2 First steps in R

R is both open-source statistical software and an object-oriented programming language widely used in a variety of academic fields.

Datasets in R can range from numerical tables to spatial data or can even be parameters defining how to plot a chart. The core concept of R is to perform operations on variables. So typing:

```
2+2
```

And pressing the Enter key will return the result:

```
[1] 4
```

In most cases, variables are handled as different *classes* of *objects*. All workflows in R thus involve defining new objects and performing analysis on these using functions. To define an object you can either use the equal sign or an arrow. For instance if we want to store a sequence of numbers in an object called *test* we can either type:

```
test <- c(1, 3, 456, 32, 1, 2, 34, 56, 54, 64)
```

or

```
test = c(1, 3, 456, 32, 1, 2, 34, 56, 54, 64)
```

Note that the 'c' character tells R to combine the sequence of numbers in the bracket, and the arrow stores the output in the object *test*. The spaces (either side of the arrow sign or either side of the equals sign or after the commas) are entirely optional. All the basic functions in R are launched by writing the name of the function followed by a string of parameters (which in many cases include other objects) within a bracket. It is important to note that the output should be stored (using the 'arrow') in a new object, otherwise the results will not be saved.

To see what is inside an object simply type the name of the object. In our case, if we type *test* and press the enter key, the outcome will be:

```
[1]   1   3 456  32   1   2  34  56  54  64
```

The number in the squared bracket indicates the number of the row. If we wish to know the class of the object we have just created, type:

```
class(test)
```

The outcome will be:

```
[1] "numeric"
```

Each number of the numeric vector *test* can be accessed separately using numerical indices. In general, each member of an object can be accessed by giving the appropriate index enclosed in square brackets. Thus if we wish to refer to the second member of object *test*, we will type the following:

```
test[2]
```

If we wish to know the number of members in a object, we will use the `length()` function as follows:

```
length(test)
```

In this specific case the outcome will be 10.

One of R's great strengths is the capacity to handle large datasets and compute complex calculations with simple commands. Calculations are usually computed for each element in the object. So if we wish to multiply the values in *test* by 2 and create a new object with the results called double.test, we simply have to type:

```
double.test <- test*2
```

We can can print out the result by typing the object name:

```
double.test
```

And the result is:

```
[1]   2   6 912  64   2   4  68 112 108 128
```

We can also do calculations involving multiple objects:

```
test.ratio <- double.test / test
test.ratio
```

## 1.3 Installing External Library Packages

The beauty of open source software packages is that they offer the possibility of integrating new sets of tools and analysis at any time, thus allowing rapid adaptation to any new requirements of the user community. R is organised as a set of *packages* or *libraries* each with its own set of functions. Slightly confusingly, the word 'package' is used when you download and install one onto your computer, but the word 'library' is used once it comes time to enable that installed package for a particular R session. At the present moment, CRAN (an R online repository that gives access to those packages that meet certain basic tests associated with ease-of-use and technical good practice) hosts more than 21800 contributed packages, most of them designed to address specific challenges in a wide range of applied statistical fields. You can get a quick, but bewilderingly vast, impression of the range of these packages by looking at the following page from CRAN, the R online archive: https://cran.r-project.org/web/views/. These add-on package can be easily installed via the following command (when your computer is connected to the Internet):

```
install.packages()
```

This will open a new window which allows you to select a CRAN mirror site. Choose the nearest one to your location and press the 'OK' button (notice that R allows you to choose CRAN mirror site only at the first invocation of `install.packages`, after that it will automatically assume the same location. To change the CRAN mirror site you should type `chooseCRANmirror()`.) Now you will see the list of all the available packages. Choose *sf* from the list and press the 'OK' button. Now you can load the specific package using the `library()` command, putting the name of the library inside the brackets. Remember that you have to load the packages you want to use every time you start R. R has an extremely large number of functions and mistyping their names (notice that R commands are case-sensitive) might be problematic. However, if you are using a Linux operating system and have Emacs, then pressing the TAB key you can easily find the names of available functions after entering the first few letters of the function you are looking for (if you are not using Linux and Emacs then ignore this). Furthermore the `help()` function can provide information on required parameters, theoretical basis and even some examples for all the R functions. To do this simply type the name `help()` and put the name of the function inside the bracket and you will be presented with a new emacs buffer containing the help text.

## 1.4 Getting Started with data.frame and matrix class objects

Most of the data we are going to handle are bivariate and multivariate, so a single numeric vector class object is not adequate for storing these. We will first explore the matrix class object (which however we will encounter rarely) and then the data.frame class object, which will be used more frequently in the following weeks.

### 1.4.1 Matrix Class Objects

1. We will first bind the three objects we have created so far as a two different types of matrix

2. Now type:

   ```
   mat1 <- rbind(test,double.test,test.ratio)
   mat2 <- cbind(test,double.test,test.ratio)
   ```

3. If you look at the contents of the newly created objects you can see the difference between them: the first one has 3 rows and 10 columns, while the second has 10 rows and three columns.

4. As with the numeric vectors, single members of a matrix can be called using numerical indices. This time, however, the index comprises two numbers, the first one indicating the row and the second one indicating the column. Thus the following line:

   ```
   mat1[2,2]
   ```

   should give you as a result 6.

5. We can also call entire rows by indicating their number and leaving a blank space for the column. For instance:

   ```
   mat1[2,]
   ```

   Which will give you the entire second row. The same principle can be applied to retrieve a column; in this case we will leave the first space blank and indicate after the comma the number of the column.

6. As with numeric vectors, basic math operations will refer to the entire matrix. Thus if we multiply a matrix by a single number the computation will be applied piecewise to each member.

### 1.4.2 Data.frame class object

Matrices and numeric vectors are able to store only one kind of data. Thus, in cases where we wish to store a mixture of numerical, logical (*i.e.* TRUE and FALSE) and character data we must use a different class of object, such as the *data.frame* class which is well-suited for handling multivariate data.

1. Now we will create a simple data.frame class object.

```
id <- seq(1,10,1)
maxlength <- c(10,12,11,13,12,12,13,12.3,11,9.5)
maxwidth <- c(20,11,30,10,40,50,23,66,23,21)
material <- c("limestone","obsidian","obsidian","flint",
  "limestone", "limestone","obsidian","flint", "obsidian","obsidian")
tools <- data.frame(id,maxlength,maxwidth,material)
tools
```

2. The last line you typed above, should have printed out the contents of your data.frame object. As with matrix class objects we can indicate specific members of the data.frame class object by using numeric indices. We can also extract entire columns using the dollar sign followed by the name of the column. Thus if we want to select *maxlength* from the data.frame we can either indicate the numeric indices:

   ```
   tools[,2]
   ```

   or use the name of the column:

   ```
   tools$maxlength
   ```

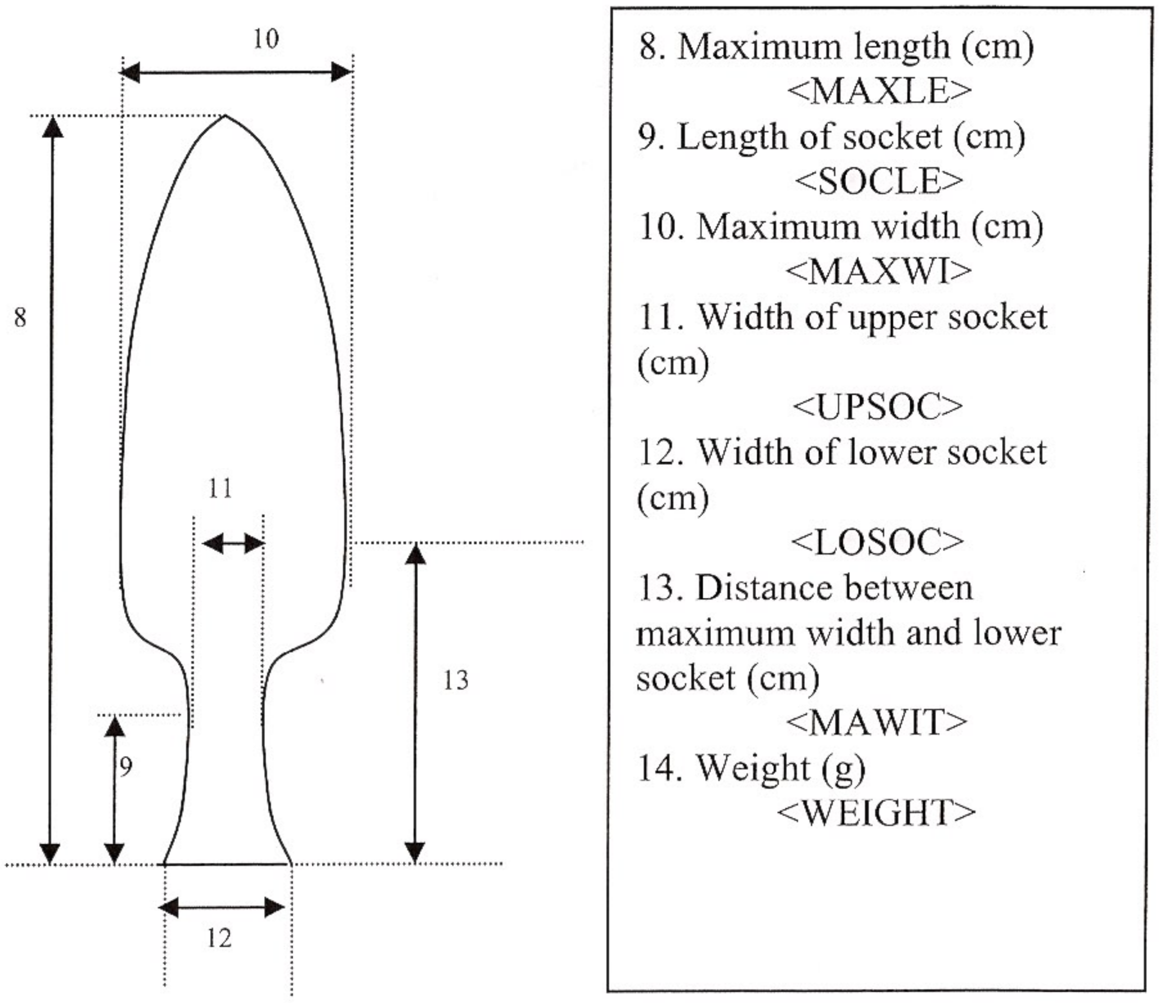
## 2 Importing and Exporting Data in R

A major strength of R is the capacity to handle extremely large datasets at a glance by using different classes of objects. However, the data creation process is typically done in external software other than R itself. We will now look at how to import both table-based (spreadsheet-like) and spatial (mappable, GIS-type) datasets.

### 2.1 Importing CSV files

Data from spreadsheet programs such Microsoft Excel or OpenOffice OpenCalc can be imported in R very easily as a data.frame class objects.

1. Make sure you have access to this week's data and have it stored in your working folder or directory for this week .
2. You should find a spreadsheet called *spearheads.xlsx.* Open it with a spreadsheet editor such as Excel (or Open Office Calc). This spreadsheet is a slightly modified version of the dataset used in the introductory manual by Fletcher and Lock (2005) and refers to series of variables associated with 40 bronze and iron spearheads from Britain.
3. The first heading (**num**) refers to the unique ID of each spearhead, while the subsequent two variables (**mat**=Material and **con**=Context) are nominal. **loo** (=Loop) and **peg** (=Peghole) are logical (TRUE or FALSE), while **cond** is the only ordinal variable (=Condition; with 1=Excellent, 2=Good, 3=Fair and 4=Poor). The other variables are numerical and are described in figure 1.
4. Now go to the menu item "File" → "Save as" and then choose in the *Save as type:* option the CSV file format and press "Save" or "OK" and save it into your folder or this week.



8. Maximum length (cm) <MAXLE>
9. Length of socket (cm) <SOCLE>
10. Maximum width (cm) <MAXWI>
11. Width of upper socket (cm) <UPSOC>
12. Width of lower socket (cm) <LOSOC>
13. Distance between maximum width and lower socket (cm) <MAWIT>
14. Weight (g) <WEIGHT>

Figure 1: Numerical variables in the spearhead dataset (Fletcher and Lock 2005: 5)

5. All work in R tends to be done within a working directory, so it is best to set this up at the beginning. In truth, for the uninitiated, it is this step of setting a working directory correctly and then working out the correct way to write the file path needed to load data that is often challenging. However, learning how to do this in a way that does not resort to a graphical user interface menu system, is really useful in the longer term, so please do persist and get help if needed.

   To set a new folder as the working directory type something like:

   ```
   setwd("X:/mycourse/week5")
   ```

6. As noted above, this act of setting the working directory takes some getting used to, and will involve you typing in a slightly different path on your machine and your operating system. If you are on a Linux machine, then it might need to be written something like ‘setwd(”/mycourse/week5’)’ or for MacOS setwd(” /mycourse/week5/”). If you have set the working directory correctly it will simply (in a very terse fashion!) return you to a

new command prompt, without error. If you have not set the working directory correctly, you will see an error and will have to try again. If you see an error, you need to try again – there is no point in continuing as nothing that follows will work correctly.

7. Setting the working environment allows us to avoid writing the path to external files every time we wish to import them. To retrieve the path of the current working directory type `getwd()`. Try this.

8. Now we will import the CSV file as a data.frame class object:

```
spearheads <- read.table(file="spearheads.csv", header=TRUE, sep=",", dec=".")
```

9. Hint: If you are using a computer with non-English language settings, you may need to adjust the 'sep' and 'dec' arguments (perhaps to ; and , respectively, but do ask for help):

10. Now we have a new data.frame class object named *spearheads*.

11. We can look the contents of *spearheads* by simply typing its name and pressing the Enter key. Try this now. Notice also that blank values in the spreadsheet have been stored as *NA*s.

12. We can now save our project in R:

```
save.image("week5.RData")
```

## 2.2 Importing Spatial Data

GIS-based spatial data can be manipulated very powerfully in an R environment thanks to a wide range of supporting library packages. In fact, there is an increasing trend for most spatial analysis specialists to do most if not all of their work in an environment such as R, using traditional GIS software only to create, organise and temporarily browse map-like datasets. We will now use a couple of supporting library packages to import GIS-type files that have been written in the well-known ESRI 'shapefile' format for vector point, line and polygon datasets.

### 2.2.1 Importing GIS Layers

1. We will first download and install the *sf* package (specifying that we want R to sort out all other related libraries and that you DO NOT want to install from source).

```
install.packages("sf", dependencies=TRUE)
```

2. We will now load it into memory:

```
library(sf)
```

3. Now we will import the polygon dataset which is provided in this week's data archive. To do so, type:

```
coast <- st_read("basic/coast/coast.shp")
```

4. This command is specifically designed to import vector files. Now if you type `coast` and press the enter key you will see a wide range of information related to the actual shapefile. To plot the shapefile type:

   ```
   plot(st_geometry(coast))
   ```

   This will open a R Graphics window with a view of the coast shapefile. Or if you wanted to give it a different colour.

   ```
   plot(st_geometry(coast), col="grey75", border="red")
   ```

5. Now save your project (as we did above with save.image(”week5.RData”)), and then quit R by typing:

   ```
   quit("no")
   ```

   where the ”no” argument means you want to quit directly with out doing any more saving of files before your quit.

## 3 A Toy Example

This tutorial now switches gear and gets a bit more complicated. It will look to introduce you briefly to several ‘point pattern analysis’ methods in R that are quite popular i archaeology. The first dataset that we will use has been artificially created in order to allow you to assess both the strength and weakness of these approaches and solve the typical pitfalls and problems which you might encounter when assessing real point distribution data. Thereafter, we look at a real dataset, but for now, we will be using the following ‘toy’ raster and vector data:

**sites** Point vector representing site locations.

**studyarea** Polygon vector map of the study area.

**elevation** 20 meter resolution DEM.

1. Now load in some relevant libraries. You may need to use install.packages() to download several of these first.

   ```
   library(sf)
   library(terra)
   library(spatstat)
   ```

2. Now load in the three datasets that are in the sub-folder ’toy’. If you get any errors, then try to troubleshoot why they may have occurred, or ask for help. You cannot proceed in an effective way if you see any errors here, so there is no point going onwards until the following data loads cleanly.

   ```
   elevation <- rast("basic/toy/elevation.asc")
   sites <- read.csv(file="basic/toy/sites.csv", header=TRUE)
   studyarea <- st_read("basic/toy/studyarea/studyarea.shp")
   ```

3. The sites data you just loaded is in fact just a spreadsheet (like the spearheads one you loaded before). we can convert it into actual spatial data, by explicitly saying which columns should be used as spatial coordinates, as follows.

   ```
   sites <- st_as_sf(sites, coords=c("X", "Y"))
   ```

4. Plot these datasets together:

   ```
   dev.new(device=pdf, height=5, width=5)
   plot(elevation)
   plot(st_geometry(studyarea), col="NA", border="red", add=TRUE)
   points(sites, pch=19, cex=0.5)
   title("A toy landscape with 20 sites")
   ```

5. Before we can start applying any useful statistical methods to our datasets, we have further to convert them into the formats preferred by the (amazingly diverse and powerful) R package ‘spatstat’.

   ```
   sta <- as.owin(studyarea)
   spts <- ppp(x=st_coordinates(sites)[,1], y=st_coordinates(sites)[,2],
   window=sta)
   ```

   Note also that in spatstat, when you create a point pattern object ‘ppp’ you are forced to define a window of analysis for it. Think how difficult it sometimes is in archaeology to say what the right ’window’ might be. Perhaps this is not a drawback of the spatstat method, but a drawback about how we have so far been thinking about archaeological distribution maps!

   Converting our raster model is harder and we have to write a custom function to do it as follows (pasting the text below in one block):

   ```
   rast2im <- function(x){
       imout <- list(v=as.matrix(x, wide=TRUE)[nrow(x):1, ],
        dim=c(nrow(x), ncol(x)),
   xrange=c(xmin(x), xmax(x)),
   yrange=c(ymin(x), ymax(x)),
   xstep=res(x),
   ystep=yres(x),
   xcol=xmin(x) + (1:ncol(x)) * xres(x) + 0.5 * xres(x),
   yrow=ymax(x) - (1:nrow(x)) * yres(x) + 0.5 * yres(x),
   type="real",
   units=list(singular=units(x), plural=units(x), multiplier=1))
       attr(imout, "class") <- "unitname"
       attr(imout, "class") <- "im"
       return(as.im(imout))
   }
   ```

6. This now allows us to the run the function to convert the raster image to a spatstat im object.

```
elev <- rast2im(elevation)
```

7. Now let's plot the results again to check we have the same data, now in a new format.

```
plot(elev, main="", col=rev(terrain.colors(100)))
plot(sta, col="NA", border="red", add=TRUE)
points(spts, pch=19, cex=0.5)
title("A toy landscape with 20 sites")
```

A key question archaeologists have been interested in asking since at least the 1960s has been: how are the points in my study area (in this case, sites in a landscape) distributed? Are they clustered, evenly distributed or random? In fact, the same question is asked of point distributions in many many other fields of research, and hence a statistical science has grown up around these issues of point pattern analysis.

8. One useful visual way to assess point distributions, especially in cases where the points are quite numerous, is to calculate a kernel density surface (or kernel density estimate, KDE). The 'bandwidth' of the spatial kernel used to summarise point density across the study area can be chosen by the user or automatically optimised (of which more below). For now we will accept a default kernel shape that is Gaussian (i.e. is shaped like a normal distribution), for which we can specify a certain standard deviation (measured in our map units, in this case metres). Put more simply, this means we either cast a large-sized spotlight (i.e. a Gaussian kernel with a high standard deviation) on the point distribution to pick out the coarse patterning, or we cast a smaller one (i.e. a kernel with a lower standard deviation) to elucidate the finer detail. To create this kernel density surface with Gaussian kernel of 500m standard deviation, and a raster cell resolution that matches the elevation dataset, type:

```
skde500 <- density(spts, sigma=500, w=as.owin(elev))
```

9. Then plot the surface and the original sites superimposed. we have chose a colour scheme which makes this look like the kinds of maps sometimes referred to more casually as a 'heatmaps'.

```
plot(skde500, col=rev(heat.colors(100)), main="Kernel density surface
of sites (sigma=500m)")
points(spts, pch=19, cex=0.5)
```

10. You will notice two or three large clusters of points on the map, one or two along the coast and one inland. Now let us repeat the same process of creating a kernel density surface, but this time choosing a different kernel size (or bandwidth), and then finally plotting both versions alongside one another:

```
skde200 <- density(spts, sigma=200, w=as.owin(elev))
dev.new(device=pdf, height=4, width=8)
par(mfrow=c(1,2))
plot(skde500, col=rev(heat.colors(100)), main="sigma=500m")
```

```
points(spts, pch=19, cex=0.5)
plot(skde200, col=rev(heat.colors(100)), main="sigma=200m")
points(spts, pch=19, cex=0.5)
par(mfrow=c(1,1))
```

What happens if you choose an even smaller bandwidth?

11. In fact, there are routines that attempt to calculate an 'optimal' bandwidth, given the nature of the point distribution. For example:

    ```
    bw.diggle(spts)
    ```

    This suggests that the most appropriate bandwidth to summarise our site data on is a standard deviation of about 150m. Occasionally such optimisation can produce results that are misleading or unhelpful, so ultimately it is up to the user to decide (and more importantly to justify their decision in some way, either because several optimisation routines agree or because there is a behavioural explanation for preferring a certain bandwidth).

## 4 Nearest Neighbour Index

1. Kernel density surfaces have offered us a visual assessment of the point pattern, but they do not provide any statistical measure of the distribution. One of the most common and traditional methods which can overcome this problem (especially in ecology) is the Clark and Evan's nearest neighbour index (which also enables a significance test). This is a method that considers the distances between each point and its nearest neighbouring point, then calculates the average of these values, and finally compares this to some measure of what we might expect these values to be in a study region of the stated size and shape. For example, we can get a full set of nearest neighbour distances for our observed sites as follows:

   ```
   nndist(spts)
   ```

   And we can plot these as a histogram or calculate the mean distance as follows:

   ```
   hist(nndist(spts))
   mean(nndist(spts))
   ```

   The average nearest neighbour distance among these sites is 273m. What the Clark And Evans index and associated significance test seek to do is suggest whether this is less than we might expect (i.e. the sites are clustered together), more than we might expect (they are regularly-spaced) or about right (they are effectively random in their spacing). We can run the test as follows:

   ```
   clarkevans.test(spts, correction="none")
   ```

2. The resulting 'R' index (sometimes also referred to as the NNI or nearest neighbour index) is about 0.89. This index is designed to indicate clustering when R<1, even distribution when R>1 and random distribution when R≃1. It can also be turned into a significance test in a variety of ways, and here we have a p-value calculated with reference to a set of theoretical expectations about the nearest neighbour distance. In our case, although the R value suggests mild clustering, the accompanying p-value suggests the pattern is not significantly different from what we might expect by chance. The latter expectation is often given the jargon term 'complete spatial randomness' or CSR.

3. Have another look at the distribution by doing a crude plot:

   ```
   dev.off()
   plot(spts)
   ```

   Does it look random? To most people, probably not... In fact, the Clark and Evans index and test (although widely used) suffer from several important problems. First, the estimate of R will often be biased due to 'edge effects', because there is a very real risk that the true nearest neighbour of a point close to the edge of the study area might actually lie outside that area. This means that observed nearest neighbour distances are likely to be larger than the true nearest neighbour distances. There are corrections for edge effects but none of them are perfect. More importantly perhaps, we have to be careful about how we choose our window of analysis (i.e. our study region). For example, if we calculate this test in a square study region that is the size of our elevation map data, instead of the irregular one based on our investigated area, we get a different outcome.

   ```
   wbox <- owin(xrange=c(xmin(elevation),xmax(elevation)), yrange=c(ymin(elevation),ymax(e
   sptsbox <- ppp(x=st_coordinates(sites)[,1], y=st_coordinates(sites)[,2],
   window=wbox)
   plot(sptsbox)
   clarkevans.test(sptsbox, correction="none")
   ```

   Here we get what appears to be a very significant result indicating strong spatial clustering of points, but of course we have included empty areas such as the sea (bottom right) that never had a chance to have sites on them. However, perhaps the most important drawback of the Clark and Evans nearest neighbour index is that provides only a single numerical index (about distance between *nearest* neighbours only) and does not consider the scale of the point process. For example, we might argue intuitively that our sites are spaced evenly and therefore 'regular' at small scales, but in contrast, cluster together into two or three parts of the map at a coarser scale. We will look at other methods that overcome this last problem below.

## 5 K functions and related methods

K functions continue the emphasis begun above on inter-point distances, but extend them by considering how many neighbouring points typically fall in circles of increasing radius around each point. The idea is to set up a set of successive buffers around every point, and then calculate, for each buffer radius, the mean number of points that falls in that buffer.

1. We can produce a first K function of out archaeological sites using the following method

   ```
   plot(Kest(spts))
   ```

   The result can look daunting because you are confronted with a result including lots of lines and symbols! Simply put, the line labelled Kpoiss(r) in dotted blue is the K function we would expect under 'complete spatial randomness' where a 'stationary Poisson process' (in the jargon) was responsible for generating the point pattern. The observed version comes to you by default in three flavours each of which involves a different way of correcting for the edge effects around your irregular study area polygon (we will not go into detail here in the practical about how each edge correction method differs, but we can discuss it in class if you want). We can plot just one kind of edge-corrected K function like this (this correction was developed for irregular polygons by the inventor of this statistical method, Brian Ripley, and is sometimes called an 'isotropic' correction):

   ```
   plot(Kest(spts, correction="iso"))
   ```

   On the plot, the K function values are plotted on the y-axis and the distance bands (the concentric buffers) are plotted on the x-axis. Note that the bands are really small.....because a K function is cumulative, you can increase the buffers in minute steps (like very small histogram bins) to get a very finely graded function. Looking at the observed K function (in black), you can see that, while they each differ a bit, the underlying pattern seems to be that they are less than we might expect at short radiuses based on the theoretical curve at short distances (i.e. fewer neighbouring points fall in buffers up to about 2-300m radius), but more a larger radiuses (more neighbouring points can be counted in buffers of larger sizes).

2. In fact, the above observation is similar to the conclusion we came to above from visual observation: the sites appear to be regularly spaced at small scales and clustered at larger ones. A further question is then whether this is really something we need to be worried about or whether than level of variation from our expectations should not be surprising. The way this is handled for K functions is via the use of Monte Carlo simulation (the logic behind which you are introduced to in tutorial 5mc).

   ```
   plot(envelope(spts, Kest, nsim=49, correction="iso"))
   ```

   The above command, does many things at once (and we will unpack it below more carefully): it calculates the same edge-corrected K function from the observed data that we were just looking at, but also then calculates 49 Monte Carlo runs. In each run, a set of random points equivalent to our observed sample size is dropped in our study area and K function calculated. The grey envelope shows the highest K values and lowest ones among all 49+1 (the 1 being the observed data) runs, creating a 'critical envelope'. If the observed data goes above this envelope, we can probably assume the pattern is a meaningful one at that scale. So, for example, the small-scale regularity is not clearly below the grey envelope (so we cannot be sure this is meaningful), but the larger-scale clustering does, so this is clearly a pattern. Now run the same thing again, but with the following adjustments (this may take a little while).

```
kf1000 <- envelope(spts, Kest, nsim=999, correction="iso", nrank=25)
plot(kf1000)
```

Note, first of all, that this time we have run the envelope and saved it to an object, then plotted that object, rather than doing it all in one command. This latter way is preferably because the object remains available to us from now on, rather than being discarded after the plot is finished. In any case, what we have done here is run 999 Monte Carlo simulations, and instead of plotting the maximum and minimum achieved K value in each distance band, we have plotted the 25th and 975th creating an envelope that encloses 95% of the random datasets. This is (not quite, but) almost the equivalent of providing a significance test at the 5% level. Again, we are able to see meaningful clustering at larger scales, but the small-scale pattern is not wholly convincing. Any idea why is the latter not more obviously significant?

3. The last task in this section is two explore two common alternatives to the K function: the L function and the pair correlation function.

```
lf1000 <- envelope(spts, Lest, nsim=999, correction="iso", nrank=25)
pcf1000 <- envelope(spts, pcf, nsim=999, correction="iso", nrank=25)
```

Let's plot these all together with the original K function.

```
dev.new(device=pdf, height=5, width=12)
par(mfrow=c(1,3))
plot(kf1000)
plot(lf1000)
plot(pcf1000, ylim=c(0,10))
par(mfrow=c(1,1))
```

These extra plots do not in this case necessarily tell us anything new, but can be seen as alternatives. The L function is merely an extra transformation of the K function values to make them plot in a linear fashion (i.e. the theoretical function in dotted red, now plots as a line). The pair correlation function can be quite useful (and produce different results from K and L functions) as it deals in *non-cumulative* buffers rather than cumulative ones.

4. So far we haven't considered the fact that the pattern we observe might be the result of a preference to some absolute properties of the landscape, rather than being the consequence of attraction or repulsion between sites. You will explore first this topic more on detail next week, but for now let's examine again our site distribution in relation to the elevation map:

```
plot(elev, main="")
plot(sta, col="NA", border="red", add=TRUE)
points(spts, pch=19, cex=0.5,col="red")
title("A toy landscape with 20 sites")
```

The site locations seem to be correlated with low elevation (darker blue) areas. We can examine this by comparing the two distributions.

We first extract the elevation values of the DEM:

```
backgroundelevation <- values(elevation, na.rm=TRUE)
```

We then extract the elevation values of the site location and visually compare with the background elevation:

```
elevationsample <- extract(elevation,sites)
boxplot(elevationsample$elevation, backgroundelevation, names=c("Site Locations","Backg
```

The two distributions clearly look different. We can further examine this with a simple k-s test:

```
ks.test(elevationsample$elevation,backgroundelevation)
```

which gives us a highly significant p-value. This leads us to question whether the clustering we observed in the K function is the result of a first order property (i.e. an induced form of spatial dependency) or it is the consequence of a genuine attraction between the sites (i.e. a second order property).

5. One way to examine this problem is to change our window of analysis so that first order effects are minimised. We will do so by creating a sub-window of analysis with a smaller range of elevation values. In this case we will explore a range between 0 and 200 meters:

```
subwindow <- elev
subwindow$v[which(subwindow$v>=200)] <- NA
subwindow <- subwindow > 0
plot(subwindow)
```

6. We can assign this new window to a copy of our *ppp* class object.

```
subwindow <- as.owin(subwindow)
spts2 <- spts
spts2$window <- subwindow
```

This enable us to compute our K function again. Notice that this time we will use another edge correction method, as Ripley's correction method cannot handle raster based windows of analysis:

```
kf500a <- envelope(spts, Kest, nsim=499, correction="trans", nrank=25)
kf500b <- envelope(spts2, Kest, nsim=499, correction="trans", nrank=25)
par(mfrow=c(1,2))
plot(kf500a,main="Original window")
plot(kf500b,main="New subwindow")
```

Do you notice any significant difference in the envelope of the two K functions? Recall that the choice of the window of analysis is part of the null hypothesis you are formulating, and consequently your results and interpretation will change accordingly.

## 6 Iron Age Coin Finds

1. We will now shift to look at a real archaeological dataset, consisting of late Iron Age coin finds from the area south-eastern England. These coins were probably minted by a particular tribe (a couple of centuries later, the Romans were calling the tribal group in this area the ‘Dobunni’, but is is hard to know whether this name can be retrojected backwards in time), and were made of either gold or silver (or perhaps in some cases a silvery copper alloy). The data comes from the UK Portable Antiquities Scheme, and I have shifted the locations and background map a little to comply with their policy about not releasing ultra-accurate spatial information, but have otherwise left the find-spots intact (they vary in locational precision, but you can probably think of them having a relative accuracy of often to the nearest 100m but sometimes to the nearest 1km). Further information on this dataset can be found in the Bevan 2012 paper in *Antiquity*.

2. We will first load and plot our coin data:

```
coins <- read.csv(file="ironagecoins/coins.csv", header=TRUE)
coins <- st_as_sf(coins, coords=c("x","y"))
seuk <- st_read("ironagecoins/seuk/seuk.shp")
dev.new(device=pdf, height=5, width=5)
plot(st_geometry(seuk), col="grey75", border="grey75")
plot(st_geometry(coins), pch=19, cex=0.3, add=TRUE)
```

3. You can see already that the distribution of coins of this perceived style, suggest something that might conceivably be a tribal territory. We can get the mean centre of this distribution by calculating the average x coordinate and average y coordinate of all find-spots.

```
mcentre <- data.frame(x=mean(st_coordinates(coins)[,1]), y=mean(st_coordinates(coins)[,
mcentre <- st_as_sf(mcentre, coords=c("x","y"))
dev.new(device=pdf, height=4, width=4)
plot(st_geometry(seuk), col="grey75", border="grey75")
plot(st_geometry(coins), pch=19, cex=0.3, add=TRUE)
plot(st_geometry(mcentre), pch=15, cex=1, col="green", add=TRUE)
```

In fact that mean centre (here in green) falls fairly close to a very large hillfort called Bagendon.

4. We shall start analysing this dataset more fully, by converting it to spatstat objects and sub-setting the coins by metal type.

```
allc <- as.ppp(st_coordinates(coins), as.owin(seuk))
au <- as.ppp(st_coordinates(coins[coins$Mat=="Au", ]),as.owin(seuk))
agcu <- as.ppp(st_coordinates(coins[coins$Mat=="AgCu", ]),as.owin(seuk))
```

5. Let’s first run a simple K function, with a crude envelope of 49 Monte Carlo simulations (this may take some time as the the study area border we are using is the coast of south-east England and it is complicated in outline!)

```
kfun1 <- envelope(allc, Kest, nsim=49)
plot(kfun1)
```

The grey coloured critical envelope for the Mote Carlo simulation is plotted under the dotted red theoretical function. Because the values of the observed K function are SO much larger than this, the envelope is a bit hard to see. This tell us that what we already knew already – that the coin distribution is extremely clustered if we look at it from the perspective of al of SE England (i.e. it probably related to a tribal territory within only part of this region). In a real analysis, it would almost not be worth doing this plot, as it is not really telling us anything new.

6. Let's have a look at a sub-region of the dataset.

```
sq <- owin(xrange=c(375000,475000), yrange=c(150000,250000))
sqallc <- allc[sq]
plot(allc)
plot(sq, border="red", add=TRUE)
plot(sqallc, col="red", add=TRUE)
```

7. The Kfunction for this will run more quickly, because the analytical window is now just a square and not a complicated coastline.

```
kfun2 <- envelope(sqallc, Kest, nsim=49)
plot(kfun2)
```

8. But it is still indicating very strong clustering, not least because coins are found only at certain sites in the landscape, and often in hoards of many coins. In fact, ordinary K functions of real (rather than hypothetical) datasets like this very often produce results that simply re-emphasise the fact that there is clustering in our data for one reason or another. There are some powerful ways to get around this problem, but we cannot cover them here in this tutorial, so for now we will switch to another way to develop some useful analysis, despite the presence of such general clustering, focusing on the comparison of gold versus silver coins.

9. First we will convert our data in to what is know as a multi-type, marked point pattern where attribute values of the points indicate whether they are one category (e.g gold) or another (e.g. silver or silver/copper mix)

```
mallc <- allc
marks(mallc) <- as.factor(coins$Mat)
plot(mallc)
```

Note the software now automatically plots this "marked point pattern" with two different symbols, triangles for gold coins and circles for silver ones.

10. We can now run a a new kind of K function that, instead of checking how many points of the same type fall in concentric buffers around a point, checks how many points of type B fall in concentric buffers around points of type A. In other words we are seeing if

gold coins cluster next to silver ones and vice versa. Note also that there are equivalent approaches for the L functions and pair correlation functions that we introduced above. This method comparing two kinds of point mark or attribute is sometimes also called a bivariate K function, and is something you will be exploring further in assessment 3.

```
bivkfun1 <- envelope(mallc, Kcross, i="AgCu", j="Au", nsim=49)
plot(bivkfun1)
```

Again, we have strong evidence of clustering as gold coins are often found very close to silver ones (e.g. at the same sites or in the same hoards). Sometimes however, the presence of one kind of archaeological find is rarely found in the presence of another kind (e.g. perhaps graves of two different types in a cemetery) and in such cases, we would find that the observed black line falls below the envelope and indicates regularity in the pattern.

11. We will end this tutorial by coming back to the utility of kernel density surfaces, but considering something a little more complicated which is a potentially very powerful way to look at archaeological datasets. Given all of the difficulties in assessing recovery bias in archaeological data (in terms of what finds survive, where in the landscape and why) , you could be forgiven for being a bit pessimistic about using certain spatial analytical approaches on undeniably patchy and partial data. One approach developed in epidemiology, is however very useful and involves a ratio-based (or more formally, case control) mapping known as a 'relative risk' surface. Such relative risk surfaces are increasingly common in assessing spatial factors effecting raw numbers of cases (e.g. of flu virus) where these can be confounded by variations in the underlying population (i.e. those potentially at risk) and/or variable recording of the necessary evidence (sounds like the problem we have as archaeologists). Put simply relative risk surfaces involve calculating the ratio of the kernel density estimation of the observed cases of a particular type to either that of another kind of case or that of the overall at-risk population.Let's calculate in our case a relative risk surface comparing the distribution of gold coins versus silver coins.

12. First we run and ordinary kernel density on all coins (note we have stipulated a specific kernel size here, but you might in different circumstances want to explore different sigma)

```
dcdens <- density(allc, sigma=7500, edge=TRUE, eps=500)
plot(dcdens)
```

13. This reminds us why we were getting such heavy clustering in our K functions...the coins are not only clustered in hoards and clustered in sites, but their are sub-regional clusters of finds as well! In any case, we can now run a relative risk surface, and clip off the result at a point where we feel the densities are too small to worry us when we starting comparing things.

```
rrdD <- relrisk(mallc, sigma=7500, edge=TRUE, eps=500)
rrdD[as.matrix(dcdens)<(0.000000004)] <- NA
```

Now create a custom heatmap for the surface and plot it, with the gold and silver coins on top.

```
tc <- colourmap(rev(heat.colors(10)), breaks=c(seq(0,1,0.1)))
plot(st_geometry(seuk), col="grey65", border="grey65")
plot(rrdD, col=tc, legend=FALSE, add=TRUE)
plot(au, pch=20, cex=0.85, col="darkred", add=TRUE)
plot(agcu, pch=22, cex=0.5, col="darkgreen", bg="cyan", lwd=0.5, add=TRUE)
plot(mcentre, pch=15, cex=1, col="black", add=TRUE)
```

14. The resulting plot suggests that the odds of finding gold coinage of 'Dobunni' type rather than silver 'Dobunni' coins are lower in the centre of the overall 'Dobunni' distribution and significantly higher on the western, Welsh side. This suggests that gold and silver coins potentially circulated in different spheres of Late Iron Age exchange. In fact, the designs on Iron Age gold coins adhered to more conservative stylistic traditions than contemporary silver ones, often suggesting linkages to kings, kingly retinues, horses and the other-worldly, and may well have been used to pay off mercenaries, bolster loyalties at tribal frontiers or pay tribute to neighbouring areas, whilst silver coins make conceivably have circulated more readily in the core of the tribal area as units of more everyday exchange.

15. You have now reached the end of the practical.