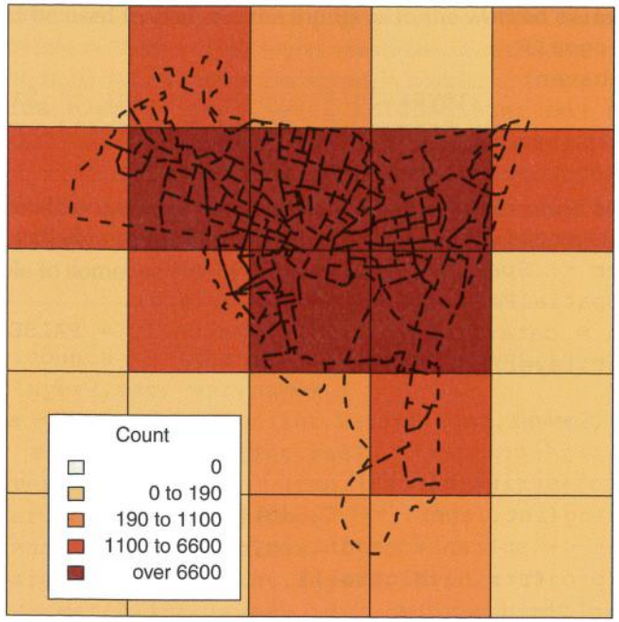


Figure 5.19 The zones shaded by population after intersection with the census blocks

REFERENCES

Bivand, R.S., Pebesma, E.J. and Gómez-Rubio, V. (2008) *Applied Spatial Data Analysis with R*. New York: Springer.

Comber, A.J., Brunsdon, C. and Green, E. (2008) Using a GIS-based network analysis to determine urban greenspace accessibility for different ethnic and religious groups. *Landscape and Urban Planning*, 86: 103–114.

Openshaw S. (1984) The modifiable areal unit problem, CATMOG 38, *Geo Abstracts*, Norwich. `http://qmrg.org.uk/files/2008/11/38-maup-openshaw.pdf`

6

POINT PATTERN ANALYSIS USING R

6.1 INTRODUCTION

In this and the next chapter, some key ideas of spatial statistics will be outlined, together with examples of statistical analysis based on these ideas, via R. The two main areas of spatial statistics that are covered are those relating to *point patterns* (this chapter) and *spatially referenced attributes* (next chapter). One of the characteristics of R, as open source software, is that R packages are contributed from a variety of authors, each using their own individual styles of programming. In particular, for point pattern analysis the `spatstat` package is often used, whilst for spatially referenced attributed, `spdep` is favoured. One the one hand `spdep` handles spatial data in the same way as `sp`, `maptools` and `GISTools`, while on the other hand `spatstat` does not. Also, for certain specific tasks, other packages may be called upon, whose mode of working differs from either of these packages. Whilst this may seem a daunting prospect, the aim of these two chapters is to introduce the key ideas of spatial statistics, as well as providing guidance in the choice of packages, and help in converting data formats. Fortunately, although some packages use different data formats, conversion is generally straightforward, and examples will appear throughout the chapters, whenever necessary.

6.2 WHAT IS SPECIAL ABOUT SPATIAL?

In one sense, the motivations for statistical analysis of spatial data are the same as those for non-spatial data:

- To explore and visualise the data;
- To create and calibrate models of the process generating the data;
- To test hypotheses related to the processes generating the data.

However, a number of these requirements are strongly influenced by the nature of spatial data. The study of mapping and cartography may be regarded as an entire subject area within the discipline of information visualisation, which focuses exclusively on geographical information.

In addition, the kinds of hypotheses one might associate with spatial data are quite distinctive – for example, focusing on the detection and location of spatial clusters of events, or on whether two kinds of event (say, two different types of crime) have the same spatial distribution. Similarly, models that are appropriate for spatial data are distinctive, in that they often have to allow for spatial autocorrelation in their random component – for example, a regression model generally include a random error term, but if the data are spatially referenced, one might expect nearby errors to be correlated. This differs from a 'standard' regression model where each error term is considered to apply independently, regardless of location. In the remainder of this section, point patterns (one of two key types of spatial data considered in this book) will be considered. Firstly, these will be described.

6.2.1 Point Patterns

Point patterns are collections of geographical points assumed to have been generated by a random process. In this case, the focus of inference and modelling is on model(s) of the random processes, and their comparison. Typically, a point dataset consists of a set of observed (x, y) coordinates, say $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where n is the number of observations. As an alternative notation, each point could be denoted by a vector $\mathbf{x}_i$, where $\mathbf{x}_i = (x_i, y_i)$. Using the data formats used in `sp`, `maptools` and so on, these data could be represented as `SpatialPoints` or `SpatialPointsDataFrame` objects. Since these data are seen as random, many models are concerned with the probability densities of the random points, $v(\mathbf{x}_i)$.

Another area of interest is the *interrelation* between the points. One way of thinking about this is to consider the probability density of one point $\mathbf{x}_i$ conditional on the remaining points $\{\mathbf{x}_1,\ldots,\mathbf{x}_{i-1},\mathbf{x}_{i+1},\ldots,\mathbf{x}_n\}$. In some situations $\mathbf{x}_i$ is independent of the other points. However, for other processes this is not the case. For example, if $\mathbf{x}_i$ is the location of the reported address for a contagious disease, then it is more likely to occur near one of the points in the dataset (due to the nature of contagion), and therefore not independent of the values of $\{\mathbf{x}_1,\ldots,\mathbf{x}_{i-1},\mathbf{x}_{i+1},\ldots,\mathbf{x}_n\}$.

Also important is the idea of a *marked process*. Here, random sets of points drawn from a number of different populations are superimposed (for example, household burglaries using force and household burglaries not using force) and the relationship between the different sets is considered. The term 'marked' is used here as the dataset can be viewed as a set of points where each point is tagged (or marked) with its parent population. Using the data formats used by `sp`, a marked process could be represented as a spatial points data frame – although the `spatstat` package uses a different format.

6.3 TECHNIQUES FOR POINT PATTERNS USING R

Having outlined the two main data types that will be considered, and the kinds of model that may be applied, in this section more specific techniques will be discussed, with examples of how they may be carried out using R. In this section, we will focus on random point patterns.

6.3.1 Kernel Density Estimates

The simplest way to consider random two-dimensional point patterns is to assume that each random location $\mathbf{x}_i$ is drawn independently from an unknown distribution with probability density function $f(\mathbf{x}_i)$. This function maps a location (represented as a two-dimensional vector) onto a probability density. If we think of locations in space as a very fine pixel grid, and assume a value of probability density is assigned to each pixel, then summing the pixels making up an arbitrary region on the map gives the probability that an event occurs in that area. It is generally more practical to assume an *unknown f*, rather than, say, a Gaussian distribution, since geographical patterns often take on fairly arbitrary shapes – for example, when applying the technique to patterns of public disorder, areas of raised risk will occur in a number of locations around a city, rather than a simplistic radial 'bell curve' centred on the city's mid-point.

A common technique used to estimate $f(\mathbf{x}_i)$ is the *kernel density estimate* or KDE (Silverman, 1986). KDEs operate by averaging a series of small 'bumps' (probability distributions in two dimensions, in fact) centred on each observed point. This is illustrated in Figure 6.1. In algebraic terms, the approximation to $f(\mathbf{x})$, for an arbitrary location $\mathbf{x} = (x, y)$, is given by

$$\hat{f}(\mathbf{x}) = \hat{f}(x,y) = \frac{1}{nh_x h_y} \sum_i k\left(\frac{x - x_i}{h_x}, \frac{y - y_i}{h_y}\right) \tag{6.1}$$

Each of the 'bumps' (central panel in Figure 6.1) map on to the kernel function $k\left(\frac{x - x_i}{h_x}, \frac{y - y_i}{h_y}\right)$ in equation (6.1) and the entire equation describes the 'bump averaging' process, leading to the estimate of probability density on the right-hand panel. Note that there are also parameters h_x and h_y (frequently referred to as the *bandwidths*) in the x and y directions; their dimension is length, and they represent the radii of the bumps in each direction. Varying h_x and h_y alters the shape of the estimated probability density surface – in brief, low values of h_x and h_y lead to very 'spiky' distribution estimates, and very high values, possibly larger than the span of the $\mathbf{x}_i$ locations, tend to 'flatten' the estimate so it appears to resemble the k-function itself; effectively this gives a superposition of nearly identical k-functions with relatively small perturbations in their centre points.

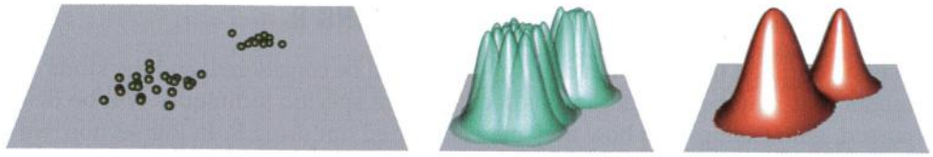


Figure 6.1 Kernel density estimation: initial points (left); bump centred on each point (centre); average of bumps giving estimate of probability density (right)

This effect of varying h_x and h_y is shown in Figure 6.2. Typically h_x and h_y take similar values. If one of these values is very different in magnitude than the other, kernels elongated in either the x or y direction result. Although this may be useful when there are strong directional effects, we will focus on the situation where values are similar for the examples discussed here. To illustrate the results of varying the bandwidths, the same set of points used in Figure 6.1 is used to provide KDEs with three different values of h_x and h_y – on the left, they both take a very low value, giving a large number of peaks; in the centre, there are two peaks; and on the right, only one.

An obvious problem is that of choosing appropriate h_x and h_y given a dataset $\{x_i\}$. There are a number of formulae to provide 'automatic' choices, as well as some more sophisticated algorithms. Here, a simple rule is used, as proposed by Bowman and Azzalini (1997) and Scott 1992:

$$h_x = \sigma_x \left(\frac{2}{3n} \right)^{\frac{1}{6}} \tag{6.2}$$

where σ_x is the standard deviation of the x_i. A similar formula exists for h_y, replacing σ_x with σ_y the standard deviation of the y_i. The central KDE in Figure 6.2 is based on choosing h_x and h_y using this method.

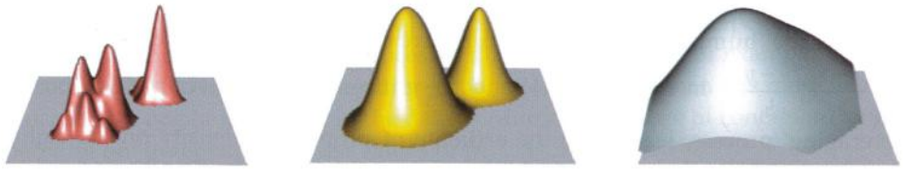


Figure 6.2 Kernel density estimation bandwidths: h_x and h_y too low (left); h_x and h_y appropriate (centre); h_x and h_y too high (right)

6.3.2 Kernel Density Estimation Using R

There are a number of packages in R that provide code for computing KDEs. Here, the `GISTools` library introduced earlier in this book will be used. The steps used to produce a KDE map here are as follows:

1. *Compute the KDE.* The function to carry out kernel density estimation is `kde.points`. This estimates the value of the density over a grid of points, and returns the result as a grid object. Here it takes two arguments – the set of points to use, and another geographical object, whose bounding box will be used to set the extent of the grid object to be created. A further optional argument allows the bandwidths to be specified. If it is omitted (as here) bandwidths are supplied via the formula in equation (6.2). Here, the breaches of the peace (public disturbances) dataset for New Haven, Connecticut, first introduced in Chapter 3, is used as an example – recall that this is provided in the `GISTools` package – here loaded using `data (newhaven)`.
2. *Plot the KDE.* Here the KDEs will be mapped as filled contour lines, rather than represented as three-dimensional objects as in Figures 6.1 and 6.2. In this way it is easier to add other geographical entities. The `level.plot` command plots the resultant grid from the KDE – although this will be seen as a rectangular grid extending beyond the New Haven study area.
3. *Clip the plot to the study area.* To clip the grid with the boundaries of the study area some further software tools are used. `poly.outer` provides a new `SpatialPolygons` object, effectively consisting of a rectangle with a hole having the shape of the second argument – a `SpatialPolygons` or `SpatialPolygonsDataFrame` object. The first argument is any kind of spatial object (in this case a spatial polygons data frame of census tracts in New Haven), and is used to determine the boundary of the rectangle. The polygon thus produced acts as a kind of 'mask', overwriting the parts of the grid that lie outside of the polygon. Finally, the `extend` argument expands this rectangular mask by the specified number of units in each direction – sometimes the default rectangle (which fits over the first argument polygon exactly) needs to be extended to cover other features of the map. The `add.masking` function then draws this object onto the map. Finally, in case some of the boundaries of the study area tracts may have been overwritten (this sometimes happens as the boundaries of the polygonal hole may coincide exactly with those of the study area), the tracts are redrawn.

The code block to perform these operations is given below, and the resultant KDE map is shown in Figure 6.3.

```
# R Kernel Density
require(GISTools)
data(newhaven)
```

```
# Compute Density
breach.dens <- kde.points(breach,lims=tracts)
# Create a level plot
level.plot(breach.dens)
# Use 'masking' to clip around blocks
masker <- poly.outer(breach.dens,tracts,extend=100)
add.masking(masker)
# Add the tracts again
plot(tracts,add=TRUE)
```

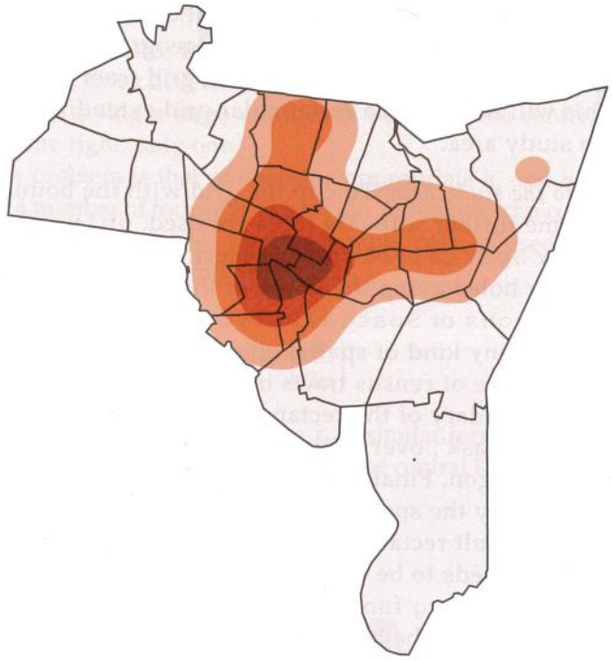


Figure 6.3 KDE map for breaches of the peace

Self-Test Question 1. As a further exercise, try adding a scale, and a layer of roads to this map. The help for `map.scale` is useful here, and an example is given in the help for the `newhaven` dataset.

6.4 FURTHER USES OF KERNEL DENSITY ESTIMATION

As well as estimating the probability density function $f(x, y)$, kernel density estimation also provides a helpful visual tool for displaying point data. Although plotting point data directly can show all of the information in a small dataset, if the dataset is larger it is hard to discriminate between relative densities of points: essentially, when points are very closely packed, the map symbols begin to overprint and exact numbers are hard to determine; this is illustrated in Figure 6.4. On the left is a plot of locations. The points plotted are drawn from a two-dimensional Gaussian distribution, and their relative density increases towards the centre. However, except for a penumbral region, the intensity of the dot pattern appears to have roughly fixed density. As the KDE estimates relative density, this problem is addressed – as may be seen in the KDE plot in Figure 6.4 (right).

Figure 6.4 The overplotting problem: point plot (left) and KDE plot (right)

KDE is also useful for comparative purposes. In the New Haven dataset there are also data relating to burglaries from residential properties. These are divided into two classes, burglaries that involve forced entry and burglaries that do not. It may be of interest to compare the spatial distributions of the two groups. In the `newhaven` dataset, `burgres.f` is a `SpatialPoints` object with points for the occurrence of forced entry residential burglaries, and `burgres.n` is a `SpatialPoints` object with points for non-forced entries. Based on the recommendation to compare patterns in data using small multiples of graphical panels (Tufte, 1990), KDE maps for forced and non-forced burglaries may be shown side

by side. This is achieved using the following block of R code, which essentially reuses the previous block, but substitutes the `SpatialPoints` object used in the KDE computations, and repeats the operation for two maps side by side. The result is seen in Figure 6.5. Although there are some similarities in the two patterns – likely due to the underlying pattern of housing – it may be seen that for the non-forced entries there is a more prominent peak in the east, whilst for forced entries the stronger peak is to the west.

```
# R Kernel Density comparison
require(GISTools)
data(newhaven)
# Set up parameters to create two plots side by side,
# with 2 line margin at the top, no margin to bottom, left
# or right
par(mfrow=c(1,2),mar=c(0,0,2,0))
# Compute density for forced entry burglaries and
create plot
brf.dens <- kde.points(burgres.f,lims=tracts)
level.plot(brf.dens)
# Use 'masking' as before
masker <- poly.outer(brf.dens,tracts,extend=100)
add.masking(masker)
plot(tracts,add=TRUE)
# Add a title
title("Forced Burglaries")

# Compute density for non-forced entry burglaries and
create plot
brn.dens <- kde.points(burgres.n,lims=tracts)
level.plot(brn.dens)
# Use 'masking' as before
masker <- poly.outer(brn.dens,tracts,extend=100)
add.masking(masker)
plot(tracts,add=TRUE)
# Add a title
title("Non-Forced Burglaries")
# reset par(mfrow)
par(mfrow=c(1,1))
```

6.4.1 Hexagonal Binning Using R

An alternative visualisation tool for geographical point datasets with larger numbers of points is *hexagonal binning*. In this approach, a regular lattice of small hexagonal cells is overlaid on the point pattern, and the number of points in each cell is counted.

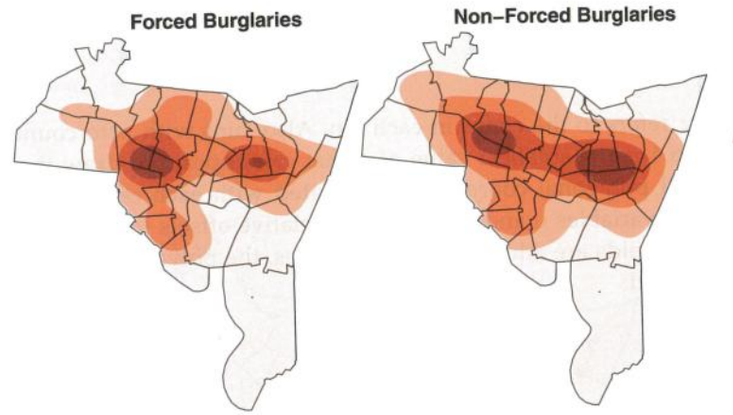


Figure 6.5 KDE maps to compare forced and non-forced burglary patterns

The cells are then shaded according to the counts. This method also overcomes the overplotting problem. However, hexagonal binning is not directly available in `GISTools`, and it is necessary to use another package. One possibility is the `fMultivar` package. This provides a routine for hexagonal binning called `hexBinning`, which takes a two-column matrix of coordinates and provides an object representing the hexagonal grid and the counts of points in each hexagonal cell. Note that this function does not work directly with `sp`-type spatial data objects. This is mainly because it is designed to apply hexagonal binning to any kind of data (for example, scatter plot points where the x and y variables are not geographical coordinates). However, it is perfectly acceptable to subject geographical points to this kind of analysis.

First, make sure that the `fMultivar` package is installed in R. If not, enter

```
install.packages("fMultivar", depend = TRUE)
```

Then use the `hexBinning` function:

```
# Load the package with hex binning
require(fMultivar)
# Create the hexagonal bins
hbins <-hexBinning(coordinates(breach))
```

The object `hbins` contains the binning information. In particular, it contains the centroid of each hexagonal bin, and the count of points in each bin:

```
head(hbins$x)
## [1] 542284 549289 550906 548212 561144 547673
head(hbins$y)
## [1] 163291 165273 165933 166594 166594 167254
```

```
head(hbins$z)
```

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

Note that z refers to the count in each bin. Also, bins with zero counts are not recorded, so 1 is the smallest value of z that can appear. To draw these on the map, the full polygonal coordinates associated with each centroid need to be entered – the variables u and v contain the relative offsets for hexagons, so that when the centroids are added to these variables, the polygons for specific polygons are created:

```
# Hex binning code block
# Set up the hexagons to plot, as polygons
u <- c(1, 0, -1, -1, 0, 1)
u <- u * min(diff(unique(sort(hbins$x))))
v <- c(1,2,1,-1,-2,-1)
v <- v * min(diff(unique(sort(hbins$y))))/3
```

Next, the background map (US Census blocks in New Haven) are drawn. Finally, the hexagons are added, here using a loop in R, so that each hexagon centre is visited in turn, and the appropriate polygon is created and shaded according to the count of points in that polygon. In this case it can be seen that there is a maximum of nine points in any polygon:

```
max(hbins$z)
```

```
## [1] 9
```

Thus, a palette of nine shades is created (via the function `brewer.pal` from the `RColorBrewer` package) is created. Note that nine is the maximum number of shades allowed for the palette used here – so if the highest value of z had exceeded 9, it would have been necessary to scale this down. Also note that at the time of writing there is no 'pre-built' function to draw hexagonal bin plots over maps, hence this code builds up this functionality from basic tools such as polygon drawing. The code to do this follows, giving the map in Figure 6.6.

```
# Draw a map with blocks
plot(blocks)

# Obtain a shading scheme
shades <-brewer.pal(9,"Greens")

# Draw each polygon, with an appropriate shade
for (i in 1:length(hbins$x)) {
```



```
  polygon(u + hbins$x[i], v + hbins$y[i],
    col = shades[hbins$z[i]],
    border = NA)}

# Re-draw the blocks
plot(blocks,add=TRUE)
```

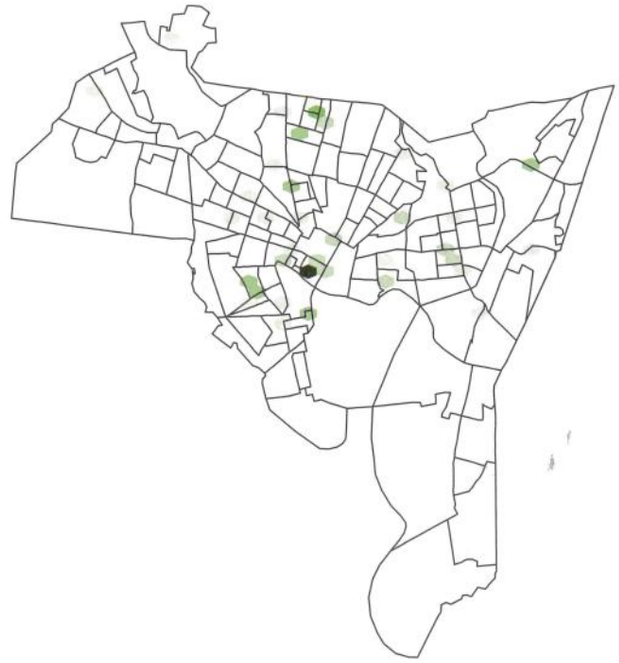


Figure 6.6 Hexagonal binning of breach of the peace events

As an alternative graphical representation, it is also possible to draw hexagons whose area is proportional to the point count. This is done by creating a variable `scaler` with which to multiply the relative polygon coordinates (this relates to the *square root* of the count in each polygon, since it is *areas* of the hexagons that should reflect the counts). This is all achieved via a modification of the previous code, listed below. The graphical output is shown in Figure 6.7.

```
# Draw a map with blocks
plot(blocks)
```

```
# Obtain a shading scheme
scaler <- sqrt(hbins$z/9)

# Draw each polygon, with an appropriate shade
for (i in 1:length(hbins$x)) {
  polygon(u*scaler[i] + hbins$x[i], v*scaler[i]
          + hbins$y[i],
    col = 'indianred',
    border = NA)}

# Re-draw the blocks
plot(blocks,add=TRUE)
```

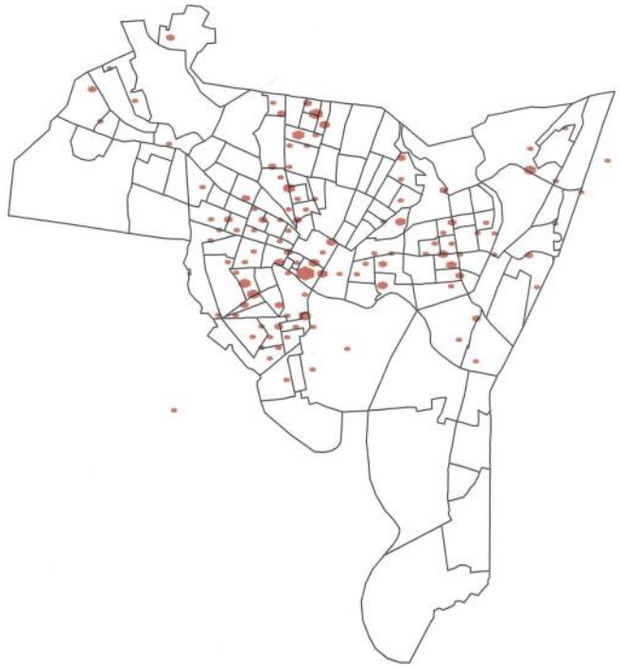


Figure 6.7 Hexagonal binning of breach of the peace events – proportional symbolism

6.5 SECOND-ORDER ANALYSIS OF POINT PATTERNS

In this section, an alternative approach to point patterns will be considered. Whereas KDEs assume that the spatial distributions for a set of points are independent but have a varying intensity, the second-order methods considered in this section assume that *marginal* distributions of points have a fixed intensity, but that

the *joint* distribution of all points is such that individual distributions of points are not independent.[1] This process describes situations in which the occurrences of events are related in some way; for example, if a disease is contagious, the reporting of an incidence in one place might well be accompanied by other reports nearby. The *K*-function (Ripley, 1981) is a very useful tool for describing processes of this kind. The *K*-function is a function of distance, defined by

$$K(d) = \lambda^{-1}\mathrm{E}(N_d) \tag{6.3}$$

where N_d is the number of events $\mathbf{x}_i$ within a distance d of a randomly chosen event from all recorded events $\{\mathbf{x}_1,\ldots,\mathbf{x}_n\}$, and λ is the intensity of the process, measured in events per unit area. Consider the situation where the distributions of $\mathbf{x}_i$ are independent, and the marginal densities are uniform – often termed a Poisson process, or *complete spatial randomness* (CSR). In this situation one would expect the number of events within a distance d of a randomly chosen event to be the intensity λ multiplied by area of a circle of radius d, so that

$$K_{\mathrm{CSR}}(d) = \pi d^2 \tag{6.4}$$

The situation in equation (6.4) can be thought of as a benchmark to assess the clustering of other processes. For a given distance d, the function value $K_{\mathrm{CSR}}(d)$ gives an indication of the expected number of events found around a randomly chosen event, under the assumption of a uniform density with each observation being distributed independently of the others. Thus for a process having a *K*-function $K(d)$, if $K(d) > K_{\mathrm{CSR}}(d)$ this suggests that there is an excess of nearby points – or, to put it another way, there is clustering at the spatial scale associated with the distance d. Similarly, if $K(d) < K_{\mathrm{CSR}}(d)$ this suggests spatial dispersion at this scale – the presence of one point suggests other points are *less* likely to appear nearby than for a Poisson process.

The consideration of spatial scale is important (many processes exhibit spatial clustering at some scales, and dispersion at others), so that the quantity $K(d) - K_{\mathrm{CSR}}(d)$ may change sign with different values of d. For example, the process illustrated in Figure 6.8 shows clustering at low values of d – for small distances (such as d_2 in the figure) there is an excess of points near to other points compared to CSR, but for intermediate distances (such as d_1 in the figure) there is an undercount of points.

When working with a sample of data points $\{\mathbf{x}_i\}$, the *K*-function for the underlying distribution will not usually be known. In this case, an estimate must be

1 A further stage in complication would be the situation where individual distributions are not independent, but also the marginal distributions vary in intensity; however, this will not be considered here.

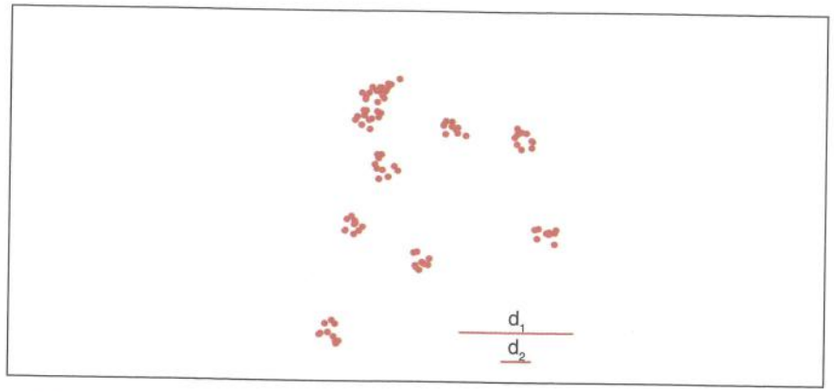


Figure 6.8 A spatial process with clustering and dispersion

made using the sample. If d_{ij} is the distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ then an estimate of $K(d)$ is given by

$$\widehat{K}(d) = \hat{\lambda}^{-1} \sum_i \sum_{j \neq i} \frac{I(d_{ij} < d)}{n(n-1)} \tag{6.5}$$

where $\hat{\lambda}$ is an estimate of the intensity – given by

$$\hat{\lambda} = \frac{n}{|A|} \tag{6.6}$$

with $|A|$ being the area of a study region defined by a polygon A. Also $I(.)$ is an indicator function taking the value 1 if the logical expression in the brackets is true, and 0 otherwise. To consider whether this sample comes from a clustered or dispersed process, it is helpful to compare $\widehat{K}(d)$ to $K_{CSR}(d)$.

Statistical inference is important here. Even if the dataset had been generated by a CSR process, an estimate of the K-function would be subject to sampling variation, and could not be expected to match $K_{CSR}(d)$ perfectly. Thus, it is necessary to test whether the sampled $\widehat{K}(d)$ is sufficiently unusual with respect to the distribution of $\widehat{K}$ estimates one might expect to see under CSR to provide evidence that the generating process for the sample is *not* CSR. The idea is illustrated in Figure 6.9. Here, 100 K-function estimates (based on equation (6.5)) from random CSR samples of 100 points (the same number of points as in Figure 6.8) are are superimposed, together with the estimate from the point set shown in Figure 6.8. From this it can be seen that the estimate from the clustered sample is quite different from the range of estimates expected from CSR.

Another aspect of sampling inference for K-functions is the dependency of $\widehat{K}(d)$ on the shape of the study area. The theoretical form $K_{CSR}(d) = \lambda\pi d^2$ is based on assumption of points occurring in an infinite two-dimensional plane. The fact that

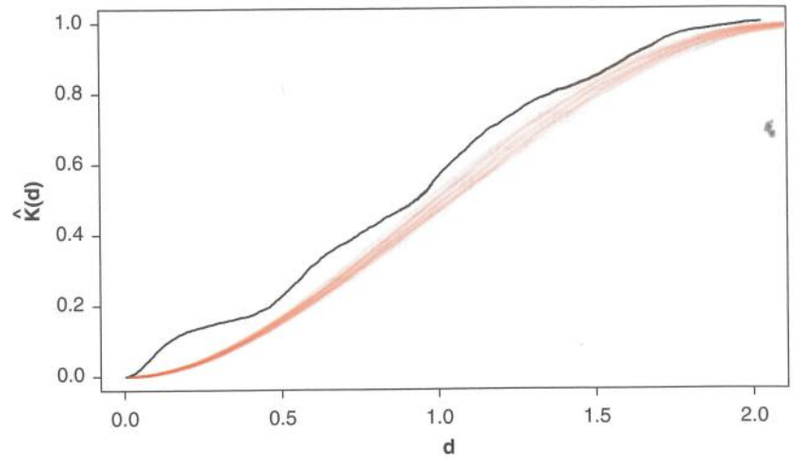


Figure 6.9 Sample K-functions under CSR

a 'real-world' sample will be taken from a finite study area (denoted here by A) will lead to further deviation of sample-based estimates of $\widehat{K}(d)$ from the theoretical form. This can also be seen in Figure 6.9 – although for the lower values of d the CSR estimated K-function curves resemble the quadratic shape expected, the curves 'flatten out' for higher values of d. This is due to the fact that for larger values of d, points will only be observed in the intersection of a circle of radius d around a random $\mathbf{x}_i$ and the study area A. This will result in fewer points being observed than the theoretical K-function would predict. This effect continues, and when d is sufficiently large any circle centred on one of the points will encompass the entirety of A. At this point, any further increase in d will result in no change in the number of points contained in the circle – this provides an explanation of the flattening-out effect seen in the figure.

Above, the idea is to consider a CSR process constrained to the study area. However another viewpoint is that the study area defines a subset of all points generated on the full two-dimensional plane. To estimate the K-function for the full-plane process some allowance for edge effects on the study area needs to be made. Ripley (1976) proposed the following modification to equation (6.5):

$$\widehat{K}(d) = \hat{\lambda}^{-1} \sum_i \sum_{j \neq i} \frac{2I(d_{ij} < d)}{n(n-1)w_{ij}} \tag{6.7}$$

where w_{ij} is the area of intersection between a circle centred at $\mathbf{x}_i$ passing through $\mathbf{x}_j$ and the study area A. Inference about the estimated K-function can then be carried out using the approach used above, but with $\widehat{K}(d)$ based on equation (6.7).

For the data in the example, points were generated with *A* as the rectangle having lower left corner (-1, -1) and upper right corner (1, 1). In practice *A* may have a more complex shape (a polygon outline of a county, for example; for this reason, assessing the sampling variability of the *K*-function under sampling must often be achieved via simulation, as seen in Figure 6.9.

6.5.1 Using the *K*-function in R

In R, a useful package for computing estimated *K*-functions (as well as other spatial statistical procedures) is `spatstat`. This is capable of carrying out the kind of simulation illustrated earlier in this section.

The *K*-function estimation as defined above may be estimated in the `spatstat` package using the `Kest` function. Here the locations of bramble canes (Hutchings, 1979; Diggle, 1983) are analysed, having been obtained as a dataset supplied with `spatstat` via the `data(bramblecanes)` command. They are plotted in Figure 6.10. Different symbols represent different ages of canes – although initially we will just consider the point pattern for *all* canes.

```
# K-function code block
# Load the spatstat package
require (spatstat)
#Obtain the bramble cane data
data (bramblecanes)
plot (bramblecanes)
```

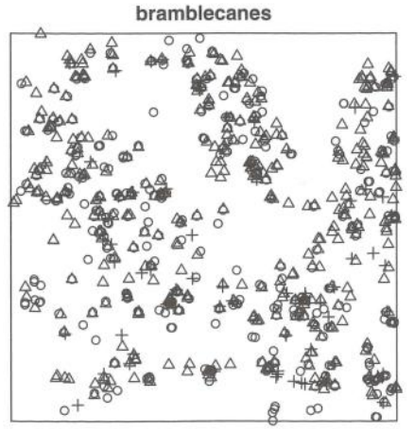


Figure 6.10 Bramble cane locations

Next the `Kest` function is used to obtain an estimate for the *K*-function of the spatial process underlying the distribution of the bramble canes. The `correction='border'` argument requests that an edge-corrected estimate (as in equation (6.7)) be used.

```
kf <- Kest(bramblecanes,correction='border')
# Plot ic
plot(kf)
```

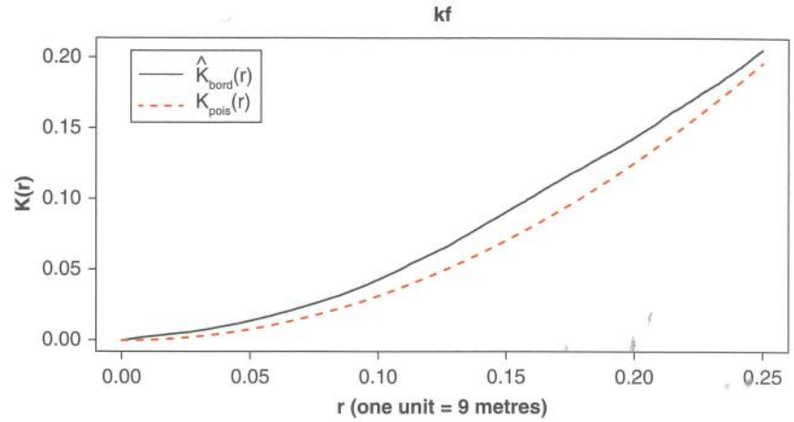


Figure 6.11 Ripley's *K*-function plot

The result of plotting the *K*-function, as shown in Figure 6.11, compares the estimated function (labelled $\hat{K}$bord) to the theoretical function under CSR (labelled $\hat{K}pois$). It may be seen that the data appear to be clustered (generally the empirical *K*-function is greater than that for CSR, suggesting that more points occur close together than would be expected under CSR). However, this perhaps needs a more rigorous investigation, allowing for sampling variation via simulation as set out above.

This simulation approach is sometimes referred to as *envelope* analysis, the envelope being the highest and lowest values of $\hat{K}(d)$ for a value of *d*. Thus the function for this is called `envelope`. This takes a `ppp` object and a further function as an argument. The function here is `Kest` – there are other functions also used to describe spatial distributions which will be discussed later, which `envelop` can use, but for now we focus on `Kest`. The envelope object may also be plotted – as shown in the following code which results in Figure 6.12:

```
# Code block to produce k-function with envelope
# Envelope function
```

```
kf.env <- envelope(bramblecanes,Kest,correction="border")
# Plot it
plot(kf.env)
```

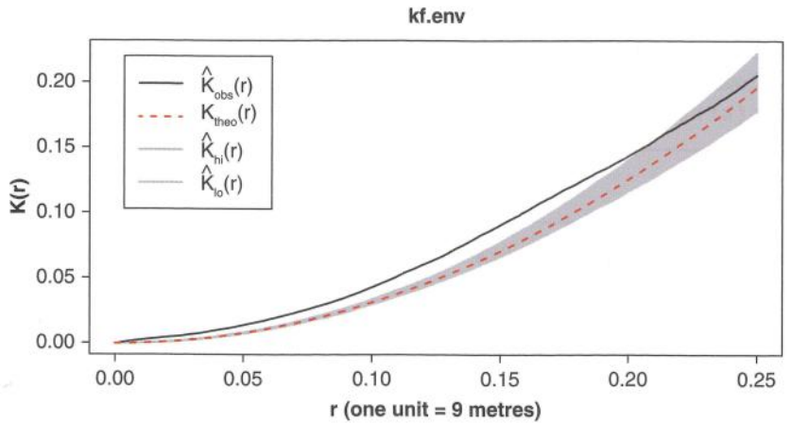


Figure 6.12 *K*-function with envelope

From this it can be seen that the estimated *K*-function for the sample takes on a higher value than the envelope of simulated *K*-functions for CSR until d becomes quite large, suggesting strong evidence that the locations of bramble canes do indeed exhibit clustering. However, it can reasonably be argued that comparing an estimated $\hat{K}(d)$ and an envelope of randomly sampled estimates under CSR is not a formal significance test. In particular, since the sample curve is compared to the envelope for several d values, multiple significance testing problems may occur. These are well explained by Bland and Altman (1995) – in short, when carrying out *several* tests, the chance of obtaining a false positive result in *any* test is raised. If the intention is to evaluate a null hypothesis of CSR, then a single number measuring departure of $\hat{K}(d)$ from $K_{CSR}(d)$, rather than the *K*-function may be more appropriate – so that a single test can be applied. One such number is the *maximum absolute deviation* (MAD: Ripley, 1977, 1981). This is the absolute value of the largest discrepancy between the two functions:

$$\text{MAD} = max_d \left| \hat{K}(d) - K_{\text{CSR}}(d) \right| \tag{6.8}$$

In R, we enter:

```
mad.test(bramblecanes,Kest)
```

```
## Generating 99 simulations of CSR ...
## 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
## 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
```



```
## 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
## 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
## 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
## 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
## 91, 92, 93, 94, 95, 96, 97, 98, 99.
##
## Done.
##
## Maximum absolute deviation test of CSR
## Monte Carlo test based on 99 simulations
## Summary function: K(r)
## Reference function: sample mean
## Interval of distance values: [0, 0.25] units (one unit = 9
## metres)
##
## data: bramblecanes
## mad = 0.016, rank = 1, p-value = 0.01
```

In this case it can be seen that the null hypotheses of CSR can be rejected at the 1% level. An alternative test is advocated by Loosmore and Ford (2006) where the test statistic is

$$u_i = \sum_{d_k=d_{min}}^{d_{max}} \left[\hat{K}_i(d_k) - \bar{K}_i(d_k) \right]^2 \delta_k \tag{6.9}$$

in which $\bar{K}_i(t_k)$ is the average value of $\hat{K}(d)$ over the simulations, the d_k are a sequence of sample distances ranging from d_{min} to d_{max} and $\delta_k = d_{k+1} - d_k$. Essentially this attempts to measure the sum of the squared distance between the functions, rather than the maximum distance. This is implemented by `spatstat` via the `dclf.test` function, which works similarly to `mad.test`.

```
dclf.test(bramblecanes,Kest)
```

```
## Generating 99 simulations of CSR ...
## 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
## 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
## 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
## 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
## 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
## 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
## 91, 92, 93, 94, 95, 96, 97, 98, 99.
##
## Done.
```

```
##
## Diggle-Cressie-Loosmore-Ford test of CSR
## Monte Carlo test based on 99 simulations
## Summary function: K(r)
## Reference function: sample mean
## Interval of distance values: [0, 0.25] units (one unit = 9
## metres)
##
## data: bramblecanes
## u = 0, rank = 1, p-value = 0.01
```

Again, results suggest rejecting the null hypothesis of CSR – see the reported *p*-value.

6.5.2 The *L*-function

An alternative to the *K*-function for identifying clustering in spatial processes is the *L*-function. This is defined in terms of the *K*-function:

$$L(d) = \sqrt{\frac{K(d)}{\pi}} \tag{6.10}$$

Although just a simple transformation of the *K*-function, its utility lies in the fact that under CSR, $L(d) = d$; that is, the *L*-function is linear, having a slope of 1 and passing through the origin. Visually identifying this in a plot of estimated *L*-functions is generally easier than identifying a quadratic function, and therefore *L*-function estimates are arguably a better visual tool. The `Lest` function provides a sample estimate of the *L*-function (by applying the transform in equation (6.10) to $\hat{K}(d)$) which can be used in place of `Kest`. As an example, recall that the `envelope` function could take alternatives to *K*-functions to create the envelope plot: in the following code, an envelope plot using *L*-functions for the bramble cane data is created (see Figure 6.13):

```
# Code block to produce k-function with envelope
# Envelope function
lf.env <- envelope(bramblecanes,Lest,correction="border")
# Plot it
plot(lf.env)
```

Similarly, it is possible to apply MAD tests or Lossmore and Ford tests using *L* instead of *K*. Again `mad.test` and `dclf.test` allow an alternative to *K*-functions to be specified. Indeed, Besag (1977) recommends using *L*-functions in place of *K*-functions in this kind of test. As an example, the following code applies the MAD test to the bramble cane data using the *L*-function.

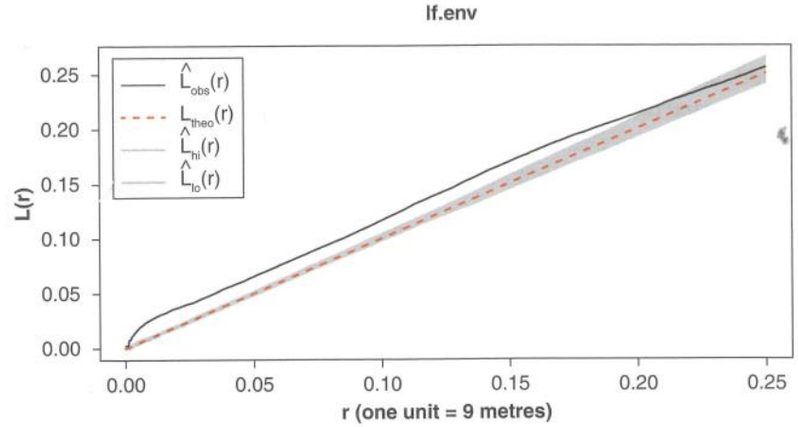


Figure 6.13 *L*-function with envelope

```
mad.test(bramblecanes,Lest)

## Generating 99 simulations of CSR ...
## 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
## 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
## 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
## 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
## 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
## 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
## 91, 92, 93, 94, 95, 96, 97, 98, 99.
##
## Done.
##
## Maximum absolute deviation test of CSR
## Monte Carlo test based on 99 simulations
## Summary function: L(r)
## Reference function: sample mean
## Interval of distance values: [0, 0.25] units (one unit = 9
## metres)
##
## data: bramblecanes
## mad = 0.0175, rank = 1, p-value = 0.01
```

6.5.3 The *G*-function

Yet another function used to describe the clustering in point patterns is the *G*-function. This is the cumulative distribution of the nearest neighbour distance for a randomly selected $\mathbf{x}_i$. Thus, given a distance d, $G(d)$ is the probability that the nearest neighbour

distance for a randomly chosen sample point is less than or equal to d. Again, this can be estimated using `spatstat`, using the function `Gest`. As in Section 6.5.2, `envelope`, `mad.test` and `dclf.test` may be used with `Gest`. Here, again with the bramble cane data, a *G*-function envelope is plotted (see Figure 6.14):

```
# Code block to produce G-function with envelope
# Envelope function
gf.env <- envelope(bramblecanes,Gest,correction="border")
# Plot it
plot(gf.env)
```

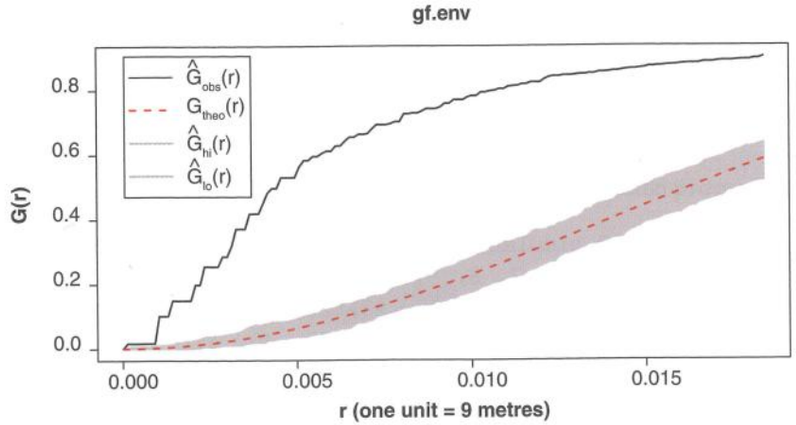


Figure 6.14 *G*-function with envelope

the estimate of the *G*-function for the sample is based on the empirical proportion of nearest neighbour distances less than d, for several values of d. In this case the envelope is the range of estimates for given d values, for samples generated under CSR. Theoretically, the expected *G*-function for CSR is

$$G(d) = 1 - \exp(-\lambda \pi d) \tag{6.11}$$

This is also plotted in Figure 6.14, as G_{theo}.

One complication is that `spatstat` stores spatial information in a different way than `sp`, `GISTools` and related packages, as noted earlier. This is not a major hurdle, but it does mean that objects of types such as `SpatialPointsDataFrame` must be converted to `spatstat`'s `ppp`

format. This is a compendium format containing both a set of points and a polygon describing the study area *A*, and can be created from a `SpatialPoints` or `SpatialPointsDataFrame` object combined with a `SpatialPolygons` or `SpatialPolygonsDataFrame` object. This is achieved via the `as` and `as.ppt` functions from the `maptools` package.

```
require (maptools)
require (spatstat)

# Bramblecanes is a data set in ppp format from
spatstat
data (bramblecanes)

# Convert the data to SpatialPoints, and plot them
bc.spformat <-as (bramblecanes,"SpatialPoints")
plot (bc.spformat)

# It is also possible to extract the study polygon
# referred to as a window in spatstat terminology
# Here it is just a rectangle...

bc.win <- as(bramblecanes$win,"SpatialPolygons" )
plot (bc.win,add=TRUE)
```

It is also possible to convert objects in the other direction, via the `as.ppp` function. This takes two arguments, the coordinates of the `SpatialPoints` or `SpatialPointsDataFrame` object (extracted using the `coordinates` function), and an `owin` object created from a `SpatialPolygons` or `SpatialPolygonsDataFrame` via `as.win`. `owin` objects are single polygons used by `spatstat` to denote study areas, and are a component of `ppp` objects. In the following example, the `burgres.n` point dataset from `GISTools` is converted to `ppp` format and a *G*-function is computed and plotted.

```
# convert burgres.n to a ppp object
br.n.ppp <- as.ppp (coordinates (burgres.n),
  W=as.owin ((gUnaryUnion (blocks))))
br.n.gf <- Gest (br.n.ppp)
plot (br.n.gf)
```

6.6 LOOKING AT MARKED POINT PATTERNS

A further advancement of the analysis of patterns of points of a single type is the consideration of *marked* point patterns. Here, several kinds of points are considered

in a dataset, instead of only a single kind. For example, in the `newhaven` dataset, there are point data for several kinds of crime. The term 'marked' is used, as each point is thought of as being tagged (or marked) with a specific type. As with the analysis of single kinds of points (or 'unmarked' points), the points are still treated as two-dimensional random quantities. It is also possible to apply tests and analyses to each individual kind of point – for example, testing each mark type against a null hypothesis of CSR, or computing the *K*-function for that mark type. However, it is also possible to examine the relationships between the point patterns of different mark types. For example, it may be of interest to determine whether forced-entry residential burglaries occur closer to non-forced-entry burglaries than one might expect if the two sets of patterns occurred independently.

One method of investigating this kind of relationship is the *cross-K-function* between marks of type i and j. This is defined as

$$K_{ij}(d) = \lambda_j^{-1} \mathrm{E}(N_{dij}) \tag{6.12}$$

where N_{dij} is the number of events $\mathbf{x}_k$ of type j within a distance d of a randomly chosen event from all recorded events $\{\mathbf{x}_1,...,\mathbf{x}_n\}$ of type i, and λ_j is the intensity of the process marked j – measured in events per unit area (Lotwick and Silverman, 1982). If the process for points with mark j is CSR, then $K_{ij}(d) = \lambda_j \pi d^2$. A similar simulation-based approach to that set out for *K*, *L* and *G* in earlier sections may be used to investigate $K_{ij}(d)$ and compare it to a hypothesised sample estimate of $K_{ij}(d)$ under CSR.

The empirical estimate of $K_{ij}(d)$ is obtained in a similar way to that in equation (6.5):

$$\widehat{K}_{ij}(d) = \hat{\lambda}_j^{-1} \sum_k \sum_l \frac{I(d_{kl} < d)}{n_i n_j} \tag{6.13}$$

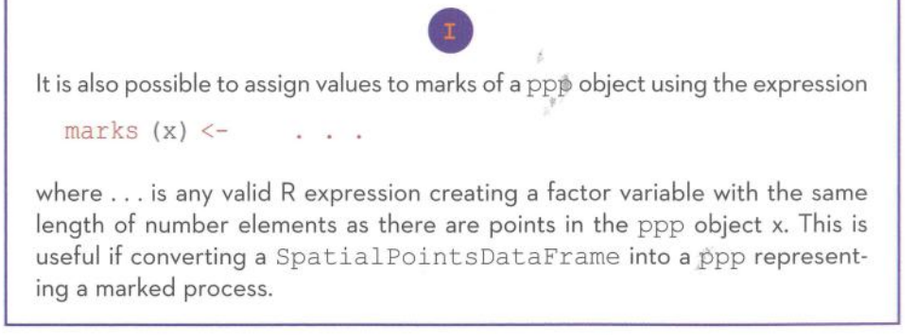
where k indexes all of the i-marked points and l indexes all of the j-marked processes, and n_i and n_j are the respective numbers of points marked i and j. A correction (of the form in equation (6.7)) may also be applied. There is also a cross-*L*-function, $L_{ij}(d)$, which relates to the cross-*K*-function in the same way that the standard *K*-function relates to the standard *L*-function.

6.6.1 Cross-*L*-function Analysis in R

There is a function in `spatstat`, called `Kcross`, to compute cross-*K*-functions, and a corresponding function called `Lcross` for cross-*L*-functions. These take a `ppp` object and values for i and j as the key arguments. Since i and j refer to mark types, it is also necessary to identify the marks for each point in a `ppp` object. This can be done via the `marks` function. For example, for the `bramblecanes` object, the points are marked in relation to the age of the cane (see Hutchings, 1979) with three levels of age (labelled as 0, 1 and 2 in increasing order). Note that the marks are factors. These may be listed by enteing:


```
marks(bramblecanes)
```

```
##   [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [71] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [106] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [141] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [176] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [211] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [281] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [351] 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [386] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [421] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [456] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [561] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [596] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [631] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [666] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [701] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [736] 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [771] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [806] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## Levels: 0 1 2
```



It is also possible to assign values to marks of a `ppp` object using the expression

```
marks (x) <-     . . .
```

where . . . is any valid R expression creating a factor variable with the same length of number elements as there are points in the `ppp` object x. This is useful if converting a `SpatialPointsDataFrame` into a `ppp` representing a marked process.

As an example here, we compute and plot the cross-*L* function for levels 0 and 1 of the `bramblecanes` object (the resultant plot is shown in Figure 6.15):

```
ck.bramble <-Lcross(bramblecanes,i=0,j=1,correction=
                    'border')
plot(ck.bramble)
```

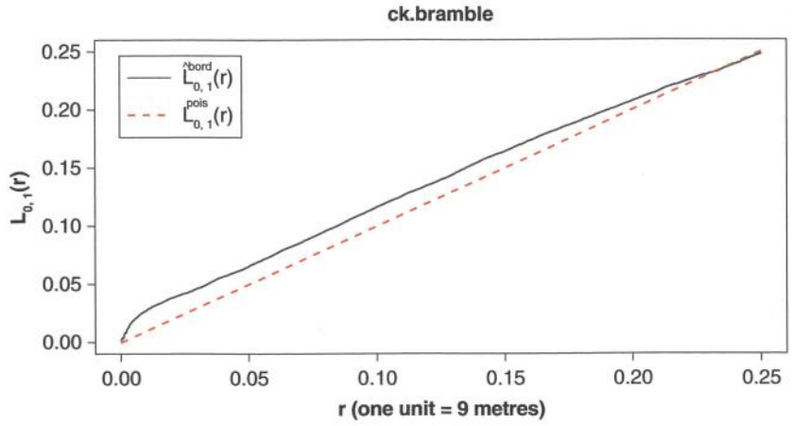


Figure 6.15 Cross-*L*-function for levels 0 and 1 of the bramble cane data

The `envelope` function may also be used (see Figure 6.16):

```
ckenv.bramble <- envelope(bramblecanes,Lcross,i=0,j=1,
correction='border')
plot(ckenv.bramble)
```

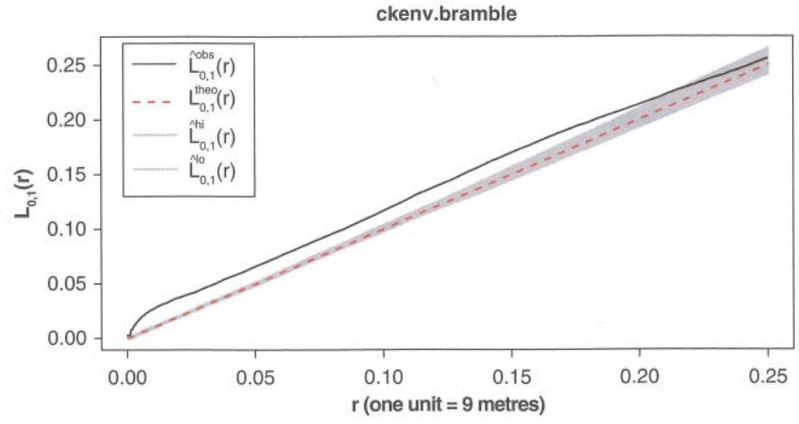


Figure 6.16 Cross-*L*-function envelope for levels 0 and 1 of the bramble cane data

Thus, it would seem that there is a tendency for more young (level 1) bramble canes to occur close to very young (level 0) canes. This can be formally tested, as both `mad.test` and `dclf.test` can be used with `Kcross` and `Lcross`. Here the use of `Lcross` with `dclf.test` is demonstrated:

```
dclf.test(bramblecanes,Lcross,i=0,j=1,correction='border')

## Generating 99 simulations of CSR ...
## 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
## 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
## 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
## 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
## 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
## 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
## 91, 92, 93, 94, 95, 96, 97, 98, 99.
##
## Done.
##
## Diggle-Cressie-Loosmore-Ford test of CSR
## Monte Carlo test based on 99 simulations
## Summary function: L["0", "1"](r)
## Reference function: sample mean
## Interval of distance values: [0, 0.25] units (one unit = 9
## metres)
##
## data: bramblecanes
## u = 0, rank = 1, p-value = 0.01
```

6.7 INTERPOLATION OF POINT PATTERNS WITH CONTINUOUS ATTRIBUTES

The previous section can be thought of as outlining methods for analysing point patterns with *categorical*-level attributes. An alternative issue is the analysis of point patterns in which the points have continuous (or measurement scale) attributes, such as height above sea level, soil conductivity or house price. A typical problem here is *interpolation*: given a sample of measurements – say $\{z_1, ..., z_n\}$ at locations $\{\mathbf{x}_1, ..., \mathbf{x}_n\}$ – the goal is to estimate the value of z at some new point $\mathbf{x}$. Possible methods for doing this can be based on fairly simple algorithms, or on more sophisticated spatial statistical models. Here, three key measures will be covered:

1. Nearest neighbour interpolation
2. Inverse distance weighting
3. Kriging

6.7.1 Nearest Neighbour Interpolation

The first of these, *nearest neighbour interpolation*, is the simplest conceptually, and can be stated as below:

- Find i such that $|\mathbf{x}_i - \mathbf{x}|$ is minimised
- The estimate of z is z_i.

In other words, to estimate z at $\mathbf{x}$, use the value of z_i at the closest observation point to $\mathbf{x}$. Since the set of closest points to $\mathbf{x}_i$ for each i form the set of Thiessen (Voronoi) polygons for the set of points, an obvious way to represent the estimates is as a set of Thiessen (Voronoi) polygons corresponding to the $\mathbf{x}_i$ points, with respective attributes of z_i. In `rgeos` there is no direct function to create Voronoi polygons, but Carson Farmer[2] has made some code available to do this, providing a function called `voronoipolygons`. This has been slightly modified by the authors, and is listed below. Note that the modified version of the code takes the points from a spatial points data frame as the basis for the Voronoi polygons on a spatial polygons data frame, and carries across the attributes of the points to become attributes of the corresponding Voronoi polygons. Thus, in effect, if the z value of interest is an attribute in the input spatial points data frame then the nearest neighbour interpolation is implicitly carried out when using this function.

The function makes use of Voronoi computation tools carried out in another package called `deldir` – however, this package does not make use of `Spatial*` object types, and therefore this function provides a 'front end' to allow its integration with the geographical information handling tools in `rgeos`, `sp` and `maptools`. Do not be too concerned if you find the code difficult to interpret – at this stage it is sufficient to understand that it serves to provide a spatial data manipulation function that is otherwise not available.

```
#
# Original code from Carson Farmer
# http://www.carsonfarmer.com/2009/09/voronoi-polygons-
with-r/
# Subject to minor stylistic modifications
#
require(deldir)
require(sp)
voronoipolygons = function(layer) {
  crds <- layer@coords
  z <- deldir(crds[,1], crds[,2])
  w <- tile.list(z)
  polys <- vector(mode='list', length=length(w))
```

2 See `http://www.carsonfarmer.com/2009/09/voronoi-polygons-with-r/`


```
for (i in seq(along=polys)) {
  pcrds <- cbind(w[[i]]$x, w[[i]]$y)
  pcrds <- rbind(pcrds, pcrds[1,])
  polys[[i]] <- Polygons(list(Polygon(pcrds)),
                         ID=as.character(i))
}
SP <- SpatialPolygons(polys)
voronoi <- SpatialPolygonsDataFrame(SP,
    data=data.frame(x=crds[,1],
    y=crds[,2],
    layer@data,
    row.names=sapply(slot(SP, 'polygons'),
      function(x) slot(x, 'ID'))))
return(voronoi)
}
```

A look at the data

Having defined this function, the next stage is to use it on a test dataset. One such dataset is provided in the `gstat` package. This package provides tools for a number of approaches to spatial interpolation – including the other two listed in this chapter. Of interest here is a data frame called `fulmar`. Details of the dataset may be obtained by entering `?fulmar` once the package `gstat` has been loaded. The data are based on airborne counts of the sea bird *Fulmaris glacialis* during August and September of 1998 and 1999, over the Dutch part of the North Sea. The counts are taken along transects corresponding to flight paths of the observation aircraft, and are transformed to densities by dividing counts by the area of observation, 0.5 km^2.

In this and the following sections you will analyse the data described above. Firstly, however, this data should be read in to R, and converted into a `Spatial*` object. The first thing you will need to do is enter the code to define the function `voronoipolygons` as listed above. The next few lines of code will read in the data (stored in the data frame `fulmar`) and then convert them into a spatial points data frame. Note that the fulmar sighting density is stored in column `fulmar` in the data frame `fulmar` – the location is specified in columns `x` and `y`. The point object is next converted into a Voronoi spatial polygons data frame to provide nearest neighbour interpolations. Having created both the point and Voronoi polygon objects, the code below then plots these (see Figure 6.17):

```
library(gstat)
library(maptools)
data(fulmar)
fulmar.spdf <- SpatialPointsDataFrame(cbind(fulmar$x,
                                      fulmar$y),
                                    fulmar)
```

```
fulmar.spdf <- fulmar.spdf[fulmar.spdf$year==1999,]
fulmar.voro <- voronoipolygons(fulmar.spdf)
par(mfrow=c(1,2),mar=c(0.1,0.1,0.1,0.1))
plot(fulmar.spdf,pch=16)
plot(fulmar.voro)

# reset par(mfrow)
par(mfrow=c(1,1))
```

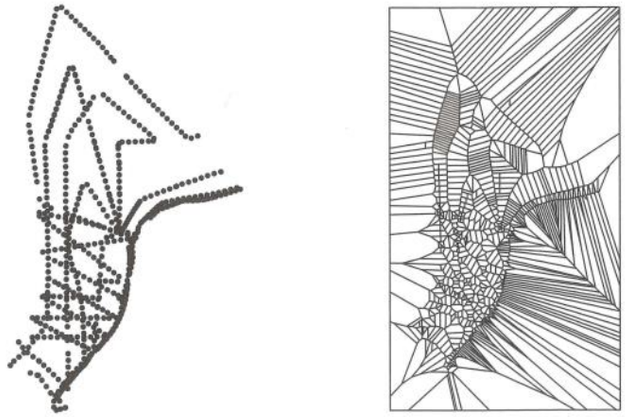


Figure 6.17 Fulmar sighting transects: (left) points; (right) Voronoi diagram

The paths of the transects become clear when the data are plotted. For the most part they are linear, although one path follows the Netherlands coast. Towards the south-west, north–south oriented paths are crossed by other zig-zag paths providing a fairly comprehensive coverage. Further north, coverage is sparser. In terms of the Voronoi diagrams, one notable artefact is that the areas of the polygons vary with the density of the points (when the points are internal) – and that edge points have polygons of infinite area (here trimmed to an enclosing rectangle). These are typical features of Voronoi polygons, but they can give rather strange characteristics to spatial interpolation. To see this, a choropleth map of the nearest neighbour densities is created. In this case, the Brewer palette `Purples` is used (higher intensity implies greater density) with break points at densities of 5, 15, 25 and 35 birds/km^2. The `par` statement controls some of the parameters used to create the plot. The `mfrow` parameter tells R to create multiple plots within the window. The plots can be thought of as a matrix, and `c(1,2)` specifies a matrix of one row and two columns – in other words, a pair of plots. The `mar` parameter specifies the margin between the drawing area in the plot and the whole area allocated. The four quantities are – in order – the bottom, left, top and right side margins in centimetres. Here these are made quite small, to allow more area to depict the map. Note that the margin area is used to add axes and titles in conventional graphs, but when working with maps with no axes, this space is not needed.

Again, some of the rather strange characteristics of the Voronoi polygon representation are apparent. In particular, the very large polygons on the edges visually dominate the interpolations somewhat, and the irregular shapes of the polygons lead to a fairly confusing visualisation. Although this approach is sometimes used as a 'quick and dirty' estimation tool (possibly as inputs to numerical models or indicators), the visual approach here does demonstrate some of the stranger characteristics of the approach. While it is possible to detect an increased density towards the north-east of the study area, it is harder to identify any subtler patterns due to the distorting effect of the variety of polygon shapes and sizes. Arguably the most problematic aspect of this approach is that the interpolated surfaces are discontinuous, and in particular that the discontinuities are an artefact of the locations of the samples. For this reason, methods such as the two others covered here are preferred. The following code results in Figure 6.18:

```
library(gstat)
library(GISTools)
sh <-shading(breaks=c(5,15,25,35),
  cols=brewer.pal(5,'Purples'))
par(mar=c(0.1,0.1,0.1,0.1))
choropleth(fulmar.voro,fulmar.voro$fulmar,shading=sh,border=NA)
plot(fulmar.voro,border='lightgray',add=TRUE,lwd=0.5)
choro.legend(px='topright',sh=sh)
```

Figure 6.18 Nearest neighbour estimate of fulmar density

6.7.2 Inverse Distance Weighting

In the *inverse distance weighting* (IDW) approach to interpolation, to estimate the value of z at location $\mathbf{x}$ a weighted mean of nearby observations is taken, rather than relying on a single nearest neighbour. To accommodate the idea that observations of z at points closer to $\mathbf{x}$ should be given more importance in the interpolation, greater weight is given to these points – in particular, if w_i is the weight given to z_i, then the estimate of z at location $\mathbf{x}$ is

$$\hat{z}(\mathbf{x}) = \frac{\sum_i w_i z_i}{\sum_i w_i} \tag{6.14}$$

where

$$w_i = |\mathbf{x} - \mathbf{x}_i|^{-\alpha} \tag{6.15}$$

and $\alpha \geq 0$. Typically $\alpha = 1$ or $\alpha = 2$, giving an inverse or inverse square relationship.

There are some interesting relationships between IDW and other methods. If $\alpha = 0$ then $w_i = 1$ for all i, and z is just the mean of all the z_i regardless of location. Also, note that the ratio of w_i to w_k, where k is the index of the closest observation to $\mathbf{x}$, is

$$\left(\frac{|\mathbf{x} - \mathbf{x}_k|}{|\mathbf{x} - \mathbf{x}_i|}\right)^{\alpha} \quad \begin{matrix} \text{if } i \neq k \\ \text{if } i = k \end{matrix} \quad \Bigg\} \tag{6.16}$$

and so if $\alpha \to \infty$ then the weighting is dominated by w_k – and the estimate of z tends to the nearest neighbour estimate.

If the value of $\mathbf{x}$ coincides with one of the $\mathbf{x}_i$ values then there is a problem with the weighting, as w_i is infinite. However, the IDW estimate is then defined to be the value of z_i. If a number (say, k) of distinct observations are all taken at the same location, so that $\mathbf{x}_{i1} = \mathbf{x}_{i2} = \ldots = \mathbf{x}_{ik}$, then the estimate is the mean of $z_{i1}, z_{i2}, \ldots, z_{ik}$.

The definition of IDW when $\mathbf{x}$ coincides with data point $\mathbf{x}_i$ is understood by noting that the IDW can be written as

$$\hat{z}(\mathbf{x}) = \frac{\sum_i d_i^{-\alpha} z_i}{\sum_i d_i^{-\alpha}}, \text{ where } d_i = |\mathbf{x} - \mathbf{x}_i| \tag{6.17}$$

and if the numerator and denominator are multiplied by d_i^{α} then

$$\hat{z}(\mathbf{x}) = \frac{z_i + d_i \sum_{k \neq i} d_k^{-\alpha} z_k}{1 + d_i \sum_{k \neq i} d_k^{-\alpha}} \tag{6.18}$$

and in the limiting case where $d_i \to 0$ this expression is just z_i as in the definition above. Also note that in the case where there are coincident locations, so that $d_{i1} = d_{i2} = \cdots = d_{ik} = d$, say then by multiplying denominator and numerator by d we have

$$\hat{z}(\mathbf{x}) = \frac{z_{i1} + z_{i2} + \ldots + z_{ik} + d\sum_{k \notin \{i_1, i_2, \ldots, i_k\}} d_k^{-\alpha} z_k}{k + d\sum_{k \notin \{i_1, i_2, \ldots, i_k\}} d_k^{-\alpha}} \tag{6.19}$$

and again, as $d \to 0$ the limit is the mean of $z_{i1}, z_{i2}, \ldots, z_{ik}$.

Computing IDW with the gstat package

There are a number of ways to compute IDW interpolation. Here, the `gstat` package will be considered. This package is also useful for kriging, the third approach to spatial interpolation covered here (Section 6.8). Thus knowledge of this package is helpful for both methods. Here, the package is demonstrated using the fulmar data used earlier. The following code carries out the IDW interpolation, and plots the interpolated surface. Firstly, you will need a set of sample points at which the IDW estimates are computed. The function `spsample` in the package `maptools` creates a sample grid.

Given a spatial polygons data frame and a number of points, if the argument `type='regular'` is provided, it will generate a spatial points data frame with a regular grid of points covering the polygon. The density of the grid such that the number of grid points is as close as possible to the number provided.[3] Here a grid with around 6000 points is created. Since the previous object `fulmar.voro` has a rectangular footprint, this causes the creation of a rectangular grid.

After this, the IDW estimate is created, using the `idw` function. This requires the model to be specified (here the formula `fulmar~1` implies that we are performing a simple interpolation) – the $\mathbf{x}_i$ locations are in `fulmar.spdf` and the points at

3 It is not always possible to find a grid with *exactly* the right number of points

which estimates are made are supplied in `s.grid`. The parameter `idp` (interpolation distance parameter) is just the value of α in the IDW – here set to 1.

```
library(maptools)  # Required package
library(GISTools)  # Required package
library(gstat)  # Set up the gstat package
# Define a sample grid then use it as a set of points
# to estimate fulmar density via IDW, with alpha=1
s.grid <- spsample(fulmar.voro,type='regular',n=6000)
idw.est <- gstat::idw(fulmar~1,fulmar.spdf,
  newdata=s.grid,idp=1.0)
```

> You may wonder why the `idw` function is referred to as `gstat::idw`. This is because an earlier package you loaded (`spatstat`) also has a function called `idw`. The notation here tells R you want to use the function in `gstat`. If you do not have `spatstat` loaded this notation is not needed - simply `idw` will do. However, if you are still in the same session where `spatstat` was used, it is difficult to guarantee which version of the function would be called. Using `gstat::idw` removes the ambiguity.

The object `idw.est` is a `SpatialPointsDataFrame` containing the IDW estimates at each of the sample points (actually a rectangular grid) in a variable called `var1.pred`. The next few lines extract the unique x and y locations in the grid, and format `var1.pred` into a matrix, `predmat`.

```
# Extract the distinct x and y coordinates of the grid
# Extract the predicted values and form into a matrix
# of gridded values
ux <- unique(coordinates(idw.est)[,1])
uy <- unique(coordinates(idw.est)[,2])
predmat <- matrix(idw.est$var1.pred,length(ux),length(uy))
```

Having created this, an alternative interpolation with $\alpha = 2$ is created using the same approach. The 'raw' IDW is stored in `idw.est2` and then stored in the matrix `predmat2`:

```
idw.est2 <- gstat::idw(fulmar~1,fulmar.spdf,
                       newdata=s.grid,idp=2.0)
predmat2 <- matrix(idw.est2$var1.pred,length(ux),length(uy))
```


Finally, both of these interpolations are mapped via the `filled.contour` function. Note that this function *adds* contours to an existing plot. Drawing the `fulmar.voro` object with colours and borders set to NA is admittedly a hack, but a useful way of creating a blank plot window with the correct extent, that the filled contour plot may be added to. Finally, although the `.filled.contour` uses a different way of specifying shading and break levels, the method from `GISTools` is used to create legends. The following code produces Figure 6.19:

```
# Draw the map. The first plot command draws nothing,

par(mar=c(0.1,0.1,0.1,0.1),mfrow=c(1,2))
plot(fulmar.voro,border=NA,col=NA)
.filled.contour(ux,uy,predmat,col=brewer.pal(5,'Purples'),
                levels=c(0,2,4,6,8,30))

# Draw the legend
sh <-shading(breaks=c(2,4,6,8),
             cols=brewer.pal(5,'Purples'))
choro.legend(px='topright',sh=sh,bg='white')

plot(fulmar.voro,border=NA,col=NA)
.filled.contour(ux,uy,predmat2,col=brewer.pal(5,'Purples'),
                levels=c(0,2,4,6,8,30))
choro.legend(px='topright',sh=sh,bg='white')
```

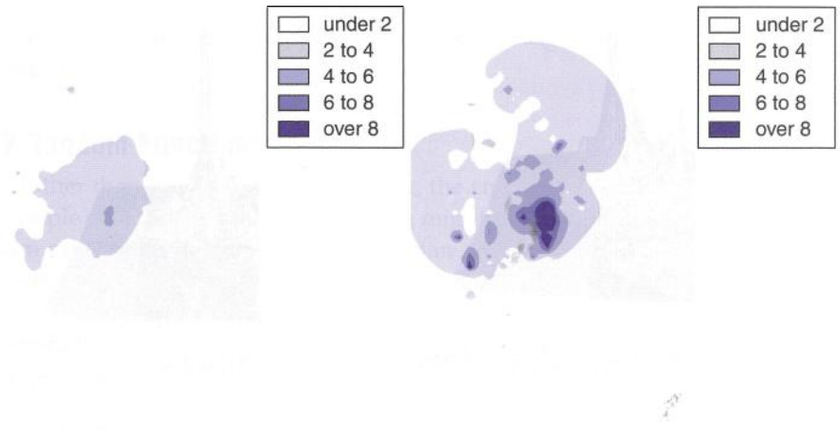


Figure 6.19 IDW Interpolation (LHS:α = 1, RHS:α = 2)

6.8 THE KRIGING APPROACH

Viewing the maps of fulmar density produced by the IDW approach, these appear to be more satisfactory than the nearest neighbour interpolation, at least in that they do consist of flat regions with a set of arbitrary linear discontinuities. However, one fact to note is that IDW interpolation always passes *exactly* through uniquely located measurement points. If the data are the result of very reliable measurement, and the underlying process is largely deterministic, this is fine. However, if the process is subject to random errors in measurement or sampling, or the underlying process is stochastic, there will be a degree of random variability in the observed z_i values – essentially z_i could be thought of as an expected 'true' value plus some random noise – so that $z_i = T(\mathbf{x}_i) + E_i$, where E_i is a random quantity with mean zero, and $T(\mathbf{x}_i)$ is a trend component. In these circumstances it is more useful to estimate the $T(\mathbf{x}_i)$ than z_i. Unfortunately, IDW interpolation does not do this. The problem here is that since this method passes through z_i it is interpolating the noise in the data as well as the trend. This is illustrated particularly well with perspective plots of the IDW interpolations. The spikes seen in the IDW surfaces for both $\alpha = 1$ and $\alpha = 2$ are a consequence of forcing the surface to go through random noise. The following R code will create these plots (see Figure 6.20):

```
par(mfrow=c(1,2),mar=c(0,0,2,0))
persp(predmat,box=FALSE)
persp(predmat2,box=FALSE)
```

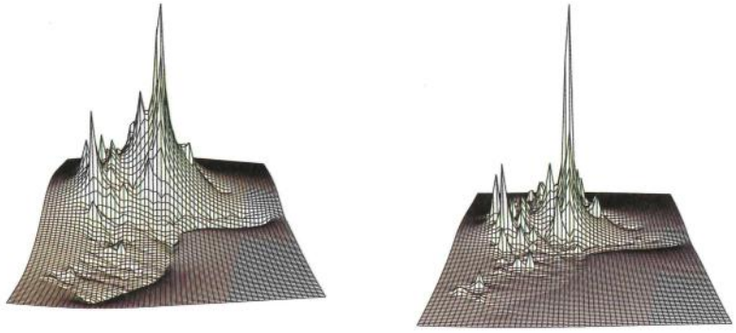


Figure 6.20 Three-dimensional plots of IDW: (left) α = 1; (right) α = 2

If multiple observations are taken at location $\mathbf{x}_i$ then the interpolated value here is the mean of the observed z values, which is a creditable estimator of $T(\mathbf{x}_i)$, but in most circumstances, only a single observation occurs at each point, and some alternative approach to interpolation should be considered. One possibility here is the use of *kriging* (Matheron, 1963). The theory behind this approach is relatively complex

(see, for example, Cressie (1993)), but a brief outline will be given. For another practical overview of the method, see Brunsdon (2009).

6.8.1 A Brief Introduction to Kriging

In kriging, the observed quantity z_i is modelled to be the outcome of a random process:

$$z_i = f(\mathbf{x}_i) + \upsilon(\mathbf{x}_i) + \varepsilon_i \tag{6.20}$$

where $f(\mathbf{x}_i)$ is a deterministic trend function, $\upsilon(\mathbf{x}_i)$ is a random function and ε_i is a random error of observation. The deterministic trend function is typical of the sorts of function often encountered in regression models – for example, a planar or quadratic function of $\mathbf{x}$ – or often just a constant mean value function. ε_i is a random variate, associated with the measurement or sampling error at the point $\mathbf{x}_i$. ε_i is assumed to have a Gaussian distribution with mean zero and variance σ^2. This is sometimes called the 'nugget' effect – kriging was initially applied in the area of mining and used to estimate mineral concentration. However, although this was modelled as a continuous quantity, in reality minerals such as gold occur in small nuggets – and exploratory mining samples taken at certain locations would be subject to highly localised variability, depending on whether or not a nugget was discovered. This effect may well be apparent in the fulmar sighting data – an observatory flight at the right time and place may spot a flock of birds, whereas one with a marginally different flight path, or slightly earlier or later may miss this.

The final term is the random function $\upsilon(\mathbf{x})$. This is perhaps the most complex to explain. If you would like to gain some further insight into this concept, read the next section. If, however, you are happy to take the kriging approach on trust, you can skip this section.

6.8.2 Random Functions

Here, rather than a single random number, the entire function is random.

A simple example of a random function might be $f(x) = a + bx$, where a and b are random numbers (say, independent Gaussian with mean zero and variances σ_a^2, σ_b^2). Since these functions are straight lines, one can think of $\upsilon(x)$ as a straight line with a random slope and intercept. For any given value of x, one could ask what the expected value of $\upsilon(x)$ is, and also what its variance is.

It is possible to derive the mean value of $\upsilon(x)$ for any value of x by noting that

$$\mathrm{E}(\upsilon(x)) = \mathrm{E}(a) + \mathrm{E}(b)x \tag{6.21}$$

and that since $\mathrm{E}(a) = \mathrm{E}(b) = 0$, $\mathrm{E}(\upsilon(x)) = 0$. This implies that if a sample of several random straight lines were generated in this way, taking their average value would give something close to zero, regardless of x. However, although the average value

of $\upsilon(x)$ may be close to zero, how might its variance change with x? Since a and b are independent,

$$\mathrm{Var}(\upsilon(x)) = \sigma_a^2 + x^2\sigma_b^2 \tag{6.22}$$

Thus, variance of the expected value of $\upsilon(x)$ increases with large absolute values of x.

Finally, suppose the function was evaluated at two values of x, say x_1 and x_2. Then some similar, but more complex, working shows that the correlation between $\upsilon(x_1)$ and $\upsilon(x_2)$ is

$$\frac{\sigma_a^2 + x_1 x_2 \sigma_b^2}{\sqrt{(\sigma_a^2 + x_1^2\sigma_b^2)\,(\sigma_a^2 + x_2^2\sigma_b^2)}} \tag{6.23}$$

Table 6.1 Some semivariogram functions

Name	Functional form
Exponential	$\gamma(d) = a(1-\exp(-d/b))$
Gaussian	$\gamma(d) = a(1-\exp(-\frac{1}{2}d^2/b^2))$
Matérn	$\gamma(d) = a\left(1-\left(2^{\kappa-1}\Gamma(\kappa)\right)^{-1}(d/b)^{\kappa}K_{\kappa}(d/b)\right)$
Spherical	$\gamma(d) = \begin{cases} a\left(\frac{3}{2}(d/b)-\frac{1}{2}(d/b)^3\right) & \text{if } d \le b \\ a & \text{otherwise} \end{cases}$

The idea here is that it is possible to define a correlation function that is related to the initial random function. It is possible, in some cases, to reverse this notion, and to define a random function in terms of the bivariate correlation function. This idea is central to kriging and geostatistics. In this case, however, a number of extensions to the above idea are applied:

- The function is defined for a vector $\mathbf{x}$ rather than a scalar x.
- Stationarity: The correlation function depends only on the distance between two vectors: say, $\rho(|\mathbf{x}_1 - \mathbf{x}_2|) = \rho(d)$ for some correlation function ρ.
- Typically the relationship is defined in terms of the *variogram*: $\gamma(d) = 2\sigma^2(1 - \rho(d))$.

- The function $\upsilon(\mathbf{x})$ is not specified directly, but deduced by 'working backwards' from $\gamma(d)$ and some observed data.

The last modification is really just convention – most practitioners of kriging specify the relationship between points in this way, rather than as a correlation or covariance. If the process is stationary, then

$$\gamma(d) = \frac{1}{2}\mathrm{E}\left[\left(\upsilon(\mathbf{x}_1) - \upsilon(\mathbf{x}_2)\right)^2\right] \tag{6.24}$$

and this can be empirically estimated from data by taking average values of the squared difference of $\upsilon(\mathbf{x}_1)$ and $\upsilon(\mathbf{x}_2)$ where the distance between $\mathbf{x}_1$ and $\mathbf{x}_2$ falls into a specified band.

Not all functional forms are valid semivariograms – however, a number of functions that are valid are well known, such as those shown in Table 6.1.

In all of these functions, the degree of correlation between $\upsilon(\mathbf{x}_1)$ and $\upsilon(\mathbf{x}_2)$ is assumed to reduce as distance increases. a and b are parameters respectively controlling the scale of variance and the extent to which nearby observations are correlated. For the Matérn semivariogram, κ is an additional shape parameter, and $K_{\kappa}(\cdot)$ is a modified Bessel function of order κ. If K = 1/2 this is equivalent to an exponential semivariogram, and as $\kappa \to \infty$ it approaches a Gaussian semivariogram.

6.8.3 Estimating the Semivariogram

As suggested earlier, equation (6.24) can be used as a way of estimating the semivariogram. Essentially all pairwise point distances are grouped into bands, and the average squared difference between $\upsilon(\mathbf{x}_1)$ and $\upsilon(\mathbf{x}_2)$ is computed for each band. Then, for one of the semivariogram functions listed above (or possibly another), a semivariogram curve is fitted – this involves finding the values of a and b that best fit the banded average squared differences described above. Note that for the Matérn case, κ is sampled at a small number of values, rather than finding a precise optimal value.

Once this is done, although $\upsilon(\mathbf{x})$ has not been explicitly calibrated, an estimate of $\gamma(d)$ is now available. In the R package `gstat` the semivariogram estimation procedure is carried out with the `variogram` function. The `boundaries` argument specifies the distance bands to work with. Here it is used with the fulmar data, and the boundaries are in steps of 5 km up to 250 km. The result of this is stored in `evgm`. Following the calibration of the estimated semivariogram `evgm`, by grouped averaging as described above, a semivariogram curve is fitted – in this case a Matérn curve. The kind of curve to fit is specified in the `vgm` function. The parameters are, in order, an estimate of a, a specification of the kind of semivariogram (`Mat` is Matérn, `Exp` is exponential, `Gau` is Gaussian and `Sph` is spherical). The next two parameters are b and κ, respectively. Note that the values provided

here are initial guesses – the `fit.variogram` function takes this specification and the `evgm` and calibrates the parameters to get a best-fit semivariogram. The result is then plotted using the `plot` function (see Figure 6.21).

```
evgm <-variogram(fulmar~1,fulmar.spdf,
                 boundaries=seq(0,250000,l=51))
fvgm <-fit.variogram(evgm,vgm(3,"Mat",100000,1))
plot(evgm,model=fvgm)
```

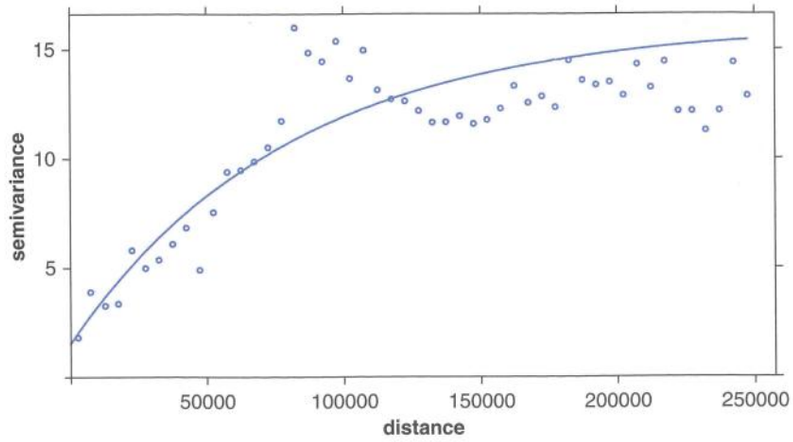


Figure 6.21 Kriging semivariogram

Once a semivariogram has been fitted it can be used to carry out the interpolation. The `fit.variogram` function estimates the 'nugget' variance discussed earlier, as well as the semivariogram parameters. Once this is done, it is possible to carry out the interpolation: essentially if a set of z_i values are available at locations $\mathbf{x}_i$ for $i = 1, \ldots, n$ and an estimate of $\gamma(d)$ is available, then $f(\mathbf{x})$ and $v(\mathbf{x})$ can be estimated for arbitrary $\mathbf{x}$ locations. Until this point, the estimation of the trend $f(\mathbf{x})$ has not been considered, but it is possible to estimate this (using more conventional regression approaches), or, if the trend is just a constant value μ, say, then the calibration of this value, and the estimation of $\mu + v(\mathbf{x})$ – essentially the interpolated value – can be carried out simultaneously using a technique termed *ordinary kriging* (see Wackernagel, 2003: 31, for example).

Operationally the interpolations are achieved by taking a weighted combination of the z_i values, $\Sigma_i w_i z_i$. In matrix form, if w_i is the weight applied to z_i, $\hat{\mu}$ is an estimate of μ, d_{ij} is the distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ and d_i is the distance between sample location $\mathbf{x}_i$ and $\mathbf{x}$, an arbitrary location at which it is desired to carry out the interpolation, then

$$\begin{bmatrix} w_1 \\ \vdots \\ w_n \\ 1 \end{bmatrix} = \begin{bmatrix} \gamma(d_{11}) & \cdots & \gamma(d_{1n}) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(d_{n1}) & \cdots & \gamma(d_{nn}) & 1 \\ 1 & \cdots & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \gamma(d_1) \\ \vdots \\ \gamma(d_n) \\ 1 \end{bmatrix} \tag{6.25}$$

However, users of `gstat` do not need to implement this, as it is made available via the `krige` function. This works in much the same way as the `idw` function, although the semivariogram model must also be supplied. This is carried out below, followed by a drawing of the fulmar density surface in the same way as before. An added bonus of kriging is that it is possible to obtain variances of the interpolated estimates as well as the estimate itself – these are derived from the statistical model – and stored in `var1.var`, alongside `var1.est`. They are useful, as they give an indication of the reliability of the estimates. Below, both the interpolated values and their variances are computed and shown in contour plots (Figure 6.22).

```
krig.est <-krige(fulmar~1,fulmar.spdf,newdata=s.grid,
                 model=fvgm)
predmat3 <-matrix(krig.est$var1.pred,length(ux),length(uy))
par(mar=c(0.1,0.1,0.1,0.1),mfrow=c(1,2))
plot(fulmar.voro,border=NA,col=NA)
filled.contour(ux,uy,pmax(predmat3,0),col=brewer.pal
               (5,'Purples'),
               levels=c(0,8,16,24,32,40))
sh <- shading(breaks=c(8,16,24,32),
              cols=brewer.pal(5,'Purples'))
choro.legend(px='topright',sh=sh,bg='white')
errmat3 <-matrix(krig.est$var1.var,length(ux),length(uy))
plot(fulmar.voro,border=NA,col=NA)
.filled.contour(ux,uy,errmat3,col=rev(brewer.pal
                (5,'Purples')),
                levels=c(0,3,6,9,12,15))
sh <- shading(breaks=c(3,6,9,12),
              cols=rev(brewer.pal(5,'Purples')))
choro.legend(px='topright',sh=sh,bg='white')
```

The plots show the interpolation and the variance. Note that on the variance map, levels are lowest (and hence reliability is highest) near to the transect flight paths – generally speaking, interpolations are at their most reliable when they are close to the observation locations.

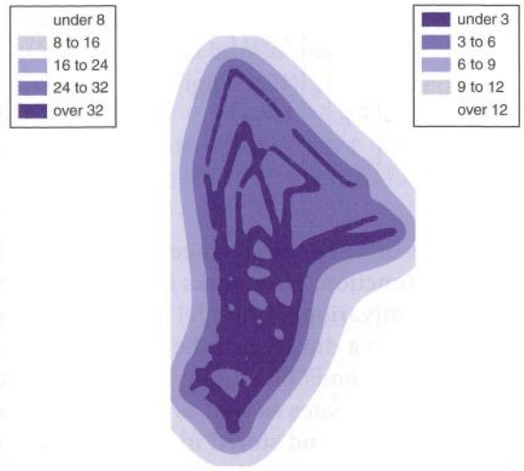


Figure 6.22 Estimates of fulmar density (right), and associated variance (left)

Finally, a perspective plot shows that although the interpolated surface is still fairly rough, some of the 'spikiness' of the IDW surface has been removed, as the surface is not forced to pass through all of the z_i (Figure 6.23).

```
persp(predmat3,box=FALSE)
```

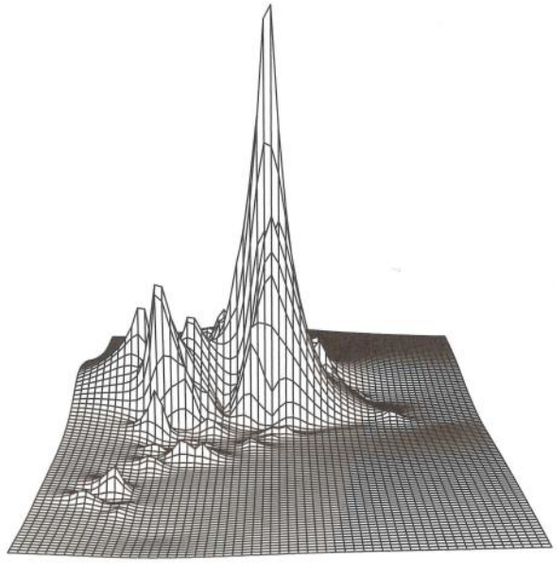


Figure 6.23 Three-dimensional plot of kriging-based interpolation

Self-Test Question 2. Try fitting an exponential variogram to the fulmar data, and creating the surface plot and maps. You may want to look at the help for `fit.variogram` to find out how to specify alternative variogram models.

6.9 CONCLUDING REMARKS

In this chapter, a number of techniques for analysing random patterns of two-dimensional points (with associated measurements in the case of interpolation), have been outlined. The key areas are first-order approaches (where the probability density function for the process is assumed to vary, and an attempt is made to estimate it) and second-order approaches (where the dependency between the spatial distributions of points is considered – this includes *K*-functions and related topics, as well as kriging). Although the chapter does not cover all possible aspects of this, it should provide an overview. As a further exercise, the reader may wish to investigate, for example, *H*-functions (Hansen et al., 1999) and their implementation in `spatstat`, or *universal kriging* (Wackernagel, 2003) where the deterministic trend function is assumed to be something more complex than a constant, as in ordinary kriging.

ANSWERS TO SELF-TEST QUESTIONS

Q1. The suggested map (Figure 6.24) could be achieved with the following code:

```
# R Kernel density
require(GISTools)
data(newhaven)
# Compute Density
breach.dens <- kde.points(breach,lims=tracts)
# Create a level plot
level.plot(breach.dens)
# Use 'masking' to clip around blocks
masker <- poly.outer(breach.dens,tracts,extend=100)
add.masking(masker)
# Add the tracts again
plot(tracts,add=TRUE)
# Add the roads
plot(roads,col='grey',add=TRUE)
# Add the scale
map.scale(534750,152000,miles2ft(2),"Miles",4,0.5,
          sfcol='red')
```

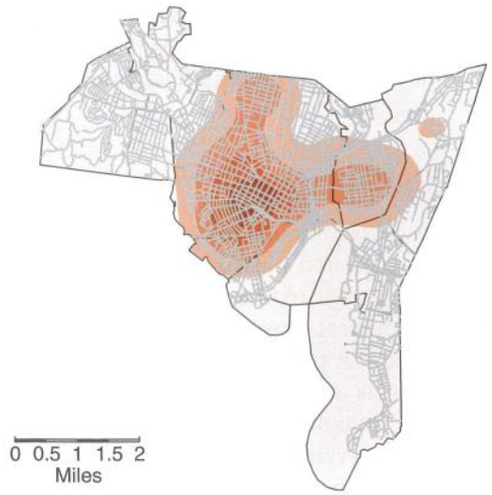


Figure 6.24 KDE map for breaches of the peace, with roads and map scale

Q2. The exponential variogram model is specified using the `"Exp"` argument in `fit.variogram` – the code to produce the variogram is given below, and the result is shown in Figure 6.25. Following this, the same procedures for producing a perspective plot or contour maps used in the above example may also be applied here.

```
evgm <- variogram(fulmar~1,fulmar.spdf,
                  boundaries=seq(0,250000,l=51))
fvgm <- fit.variogram(evgm,vgm(3,"Exp",100000,1))
plot(evgm,model=fvgm)
```

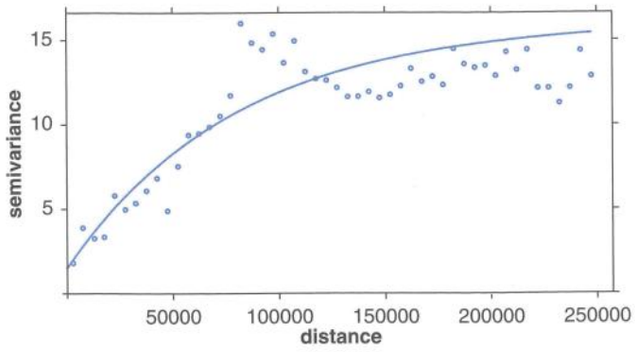


Figure 6.25 Kriging semivariogram (exponential model)

REFERENCES

Besag, J. (1977) Discussion of Dr. Ripley's paper. *Journal of the Royal Statistical Society, Series B*, 39: 193–195.

Bland, J.M. and Altman, D.G. (1995) Multiple significance tests: The Bonferroni method. *British Medical Journal*, 310: 170.

Bowman, A. and Azzalini, A. (1997) *Applied Smoothing Techniques for Data Analysis: The Kernel Approach with S-Plus Illustrations*. Oxford: Oxford University Press.

Brunsdon, C. (2009) Geostatistical analysis of lidar data. In G. Heritage and A. Large (eds), *Laser Scanning for the Environmental Sciences*. Chichester: Wiley-Blackwell.

Cressie, N. (1993) *Statistics for Spatial Data*. New York: John Wiley & Sons.

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

Hansen, M., Baddeley, A. and Gill, R. (1999) First contact distributions for spatial patterns: regularity and estimation. *Advances in Applied Probability*, 31: 15–33.

Hutchings, M. (1979) Standing crop and pattern in pure stands of *Mercurialis perennis and Rubus fruticosus* in mixed deciduous woodland. *Oikos*, 31: 351–357.

Loosmore, N.B. and Ford, E.D. (2006) Statistical inference using the *G* or *K* point pattern spatial statistics. *Ecology*, 87(8): 1925–1931.

Lotwick, H.W. and Silverman, B.W. (1982) Methods for analysing spatial processes of several types of points. *Journal of the Royal Statistical Society, Series B*, 44(3): 406–413.

Matheron, G. (1963) Principles of geostatistics. *Economic Geology*, 58: 1246–1266.

Ripley, B.D. (1976) The second-order analysis of stationary point processes. *Journal of Applied Probability*, 13: 255–266.

Ripley, B.D. (1977) Modelling spatial patterns (with discussion). *Journal of the Royal Statistical Society, Series B*, 39: 172–212.

Ripley, B.D. (1981) *Spatial Statistics*. New York: John Wiley & Sons.

Scott, D. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization*. New York: John Wiley & Sons.

Silverman, B.W. (1986) *Density Estimation for Statistics and Data Analysis*. London: Chapman & Hall.

Tufte, E.R. (1990) *Envisioning Information*. Cheshire, CT: Graphics Press.

Wackernagel, H. (2003) *Multivariate Geostatistics*. Berlin: Springer.