5 USING R AS A GIS

5.1 INTRODUCTION

In GIS and spatial analysis, we are often interested in finding out how the information contained in one spatial dataset relates to that contained in another. The kinds of questions we may be interested in include:

- How does X interact with Y?
- How many X are there in different locations of Y
- How does the incidence of X relate to the rate of Y?
- How many of X are found within a certain distance of Y?
- How does process X vary with Y spatially?

X and Y may be diseases, crimes, pollution events, attributed census areas, environmental factors, deprivation indices or any other geographic process or phenomenon that you are interested in understanding. Answering such questions using a spatial analysis frequently requires some initial data pre-processing and manipulation. This might be to ensure that different data have the same spatial extent, describe processes in a consistent way (for example, to compare land cover types from different classifications), are summarised over the same spatial framework (for example, census reporting areas), are of the same format (raster, vector, etc.) and are projected in the same way (the latter was introduced in Chapter 3).

This chapter uses worked examples to illustrate a number of fundamental and commonly applied spatial operations on spatial datasets. Many of these form the basis of most GIS software. The datasets may be ones you have read into R from shapefiles or ones that you have created in the course of your analysis. Essentially, the operations illustrate different methods for extracting information from one spatial dataset based on the spatial extent of another. Many of these are what are frequently referred to as *overlay* operations in GIS software such as ArcGIS or QGIS, but here are extended to include a number of other types of data manipulation. The sections below describe the following operations:

- Intersections to clip one dataset to the extent of another
- Creating buffers around features
- Merging the features in a spatial dataset
- Point-in-polygon and area calculations
- Creating distance attributes
- Combining spatial data and attributes
- Converting between raster and vector

As you work through the example code in this chapter a number of self-test questions are introduced. Some of these go into much greater detail and complexity than in earlier chapters and are accompanied with extensive direction for you to work through and follow.

The `GISTools` and `rgeos` packages have a number of functions for performing overlay and other spatial operations on spatial datasets which create new data, information or attributes. In many cases, it is up to the analyst (you!) to decide the order of operations in a particular analysis and, depending on your objectives, a given operation may be considered as a pre-processing step or as an analytical one. For example, calculating distances, areas, or point-in-polygon counts prior to a statistical test may be pre-processing steps prior to the actual data analysis or used as the actual analysis itself. The key feature of these operations is that they create new data or information.

5.2 SPATIAL INTERSECTION OR CLIP OPERATIONS

The GISTools package comes with dataset describing tornados in the USA. Load the package and this data into a new R session:

```
library(GISTools)
data(tornados)
```

You will see that four datasets are now loaded: `torn`, `torn2`, `us_states` and `us_states2`. The `torn` and `torn2` data describe the locations of tornados recorded between 1950 and 2004, and the `us_states` and `us_states2` datasets are spatial data describing the states of the USA. Two of these are in WGS84 projections (`torn` and `us_states`) and two are projected in a GRS80 datum (`torn2` and `us_states2`).

We can plot these and examine the data as in Figure 5.1.

```
# set plot parameters and initial plot for map extent
par(mar=c(0,0,0,0))
plot(us_states)
```

```
# plot the data using a shading with a transparency term
# see the add.alpha() function for this
plot(torn, add = T, pch = 1, col = "#FB6A4A4C", cex = 0.4)
plot(us_states, add = T)
```

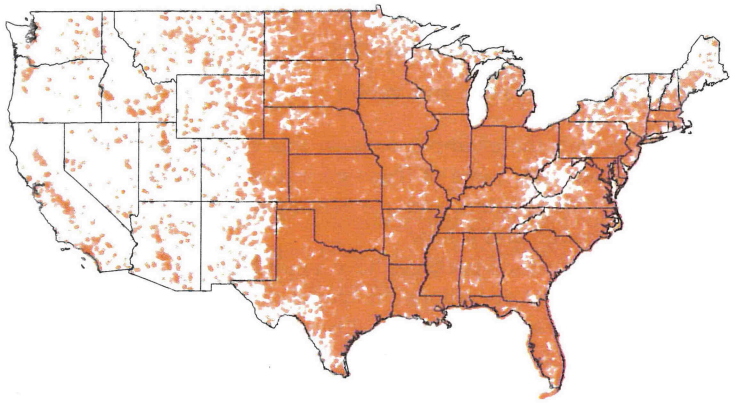

Figure 5.1 The tornado data

Remember that you can examine the attributes of a variable using the `summary()` function. So, for example, to see the projection and attributes of `torn`, enter:

```
summary(torn)
```

Now, consider the situation where the aim was to analyse the incidence of tornados in a particular area: we do not want to analyse *all* of the tornado data but only those records that describe events in our study area – the area we are interested in. The code below selects a group of US states, in this case Texas, New Mexico, Oklahoma and Arkansas – note the use of the OR logical operator `'|'` to make the selection – and then plots the tornado data over that.''

```
index <- us_states$STATE_NAME == "Texas" |
    us_states$STATE_NAME == "New Mexico" |
    us_states$STATE_NAME == "Oklahoma" |
    us_states$STATE_NAME == "Arkansas"
AoI <- us_states[index,]
```

This can be plotted using the usual commands as in the code below. You can see that the plot extent is defined by the spatial extent of area of interest (called `AoI`) and that all of the tornados within that extent are displayed.


```
plot(AoI)
plot(torn, add = T, pch = 1, col = "#FB6A4A4C")
```

However, it is possible to select *only* those records from the tornado data that are within the area we are interested in using a spatial intersection (sometimes referred to as a clip operation in GIS software such as ArcGIS). The `gIntersection` function allows us to do this as shown in the code below. The results are mapped in Figure 5.2:

```
AoI.torn <- gIntersection(AoI, torn)
par(mar=c(0,0,0,0))
plot(AoI)
plot(AoI.torn, add = T, pch = 1, col = "#FB6A4A4C")
```

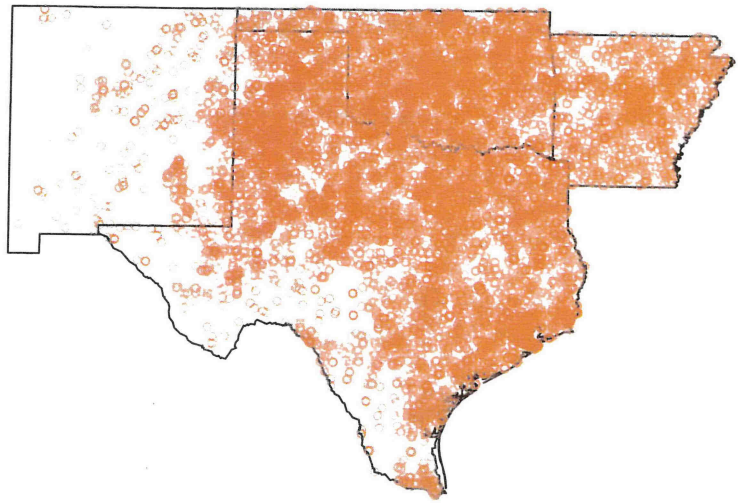

Figure 5.2 The tornado data in the defined area of interest

The `gIntersection` operation creates a `SpatialPoints` dataset of the locations of the tornados within the area of interest. However, if you examine the data created by the intersection, you will notice that it has lost its attributes: it has no `data.frame` and if you examine the first few rows of the `AoI.torn` object, by entering `head(AoI.torn)` at the R prompt, it returns only a list of coordinates.

To preserve the data attributes, the `gIntersection` command needs to be modified so that the results include the data attributes and not just their locations. This is done by including a parameter in the call to `gIntersection` to include object IDs. This takes slightly longer to run, but the variable that is created as a

result of the operation contains references to the data frames of both the input spatial objects:

```
AoI.torn <- gIntersection(AoI, torn, byid = TRUE)
```

You can examine the attributes of the `AoI.torn` data by entering:

```
head(data.frame(AoI.torn))
head(rownames(data.frame(AoI.torn)))
tail(rownames(data.frame(AoI.torn)))
```

You will notice that the intersection object, `AoI.torn` has `rownames` that indicate the origins of each tornado point: they are a composite of the row names of both inputs. In this case the row names of the `us_states` object are from 1 to 50. The ones we are interested in can be extracted using the `index` variable created above:

```
rownames(data.frame(us_states[index,]))

## [1] "37" "40" "41" "46"

us_states$STATE_NAME[index]

## [1] Oklahoma  Texas  New Mexico Arkansas
## 51 Levels: Alabama Alaska Arizona Arkansas ... Wyoming
```

The `rownames` of `AoI.torn` can be used to extract the data from `us_states` and/or `torn`. In the examples below, first the tornado attributes and then the state in which the tornado occurred are extracted and then attached as attributes to the intersected data. These operations are used to create two `data.frame` variables `df1` and `df2` which are then combined using the `cbind` function.

To extract the tornado attributes from the `torn` data frame the `strsplit` function can be used to separate the `rownames` of the intersected data into references that relate to the intersection inputs. Note that another method for *splitting strings* is given in the box below using the `gsub` function. Then `as.numeric` is used to coerce the character vectors to numbers which are then used as an index to extract the data from the `torn` data frame:

```
# assign rownames to tmp and split the data by spaces " "
tmp <- rownames(data.frame(AoI.torn))
tmp <- strsplit(tmp, " ")
# assign the first and second parts of the split
```



```
torn.id <- (sapply(tmp, "[[", 2))
state.id <- (sapply(tmp, "[[", 1))
# use torn.id to subset the torn data and assign to df1
torn.id <- as.numeric(torn.id)
df1 <- data.frame(torn[torn.id,])
```

The `state.id` and `torn.id` can be used to link to each input data frame. At the end of these operations the variable `df1` contains the information from the `torn data.frame` for each of the data points in the area of interest.

The `strsplit` function above is a convenient way for extracting the required information from character variables or *strings*. Another useful function is `gsub`, as in the code below. Notice the use of the space in the `replace.val` variable when it is defined using `sprintf: "%s"` to replace the unwanted text in the `rownames` of the `AoI.torn` character vector (in this case references to the US state data). In the code below, the elements of the variable `tmp` are reassigned or overwritten by the output of each iteration of the loop:

```
# set up some variables
state.list <- rownames(data.frame(us_states[index,]))
tmp <- rownames(data.frame(AoI.torn))
# loop through these, removing the state.list variable
for (i in 1: length(state.list)) {
    replace.val <- sprintf("%s ", state.list[i])
    tmp <- gsub(replace.val, " ", tmp)
}
# again use torn.id to subset the torn data and assign to df1
torn.id <- as.numeric(tmp)
df1 <- data.frame(torn[torn.id,])
```

To extract the state names for each tornado, the `state.id` can be used to create a second temporary variable `df2`. The two temporary data variables, `df1` and `df2`, are joined together using `cbind` and assigned to a variable called `df`, the final data frame:

```
df2 <- us_states$STATE_NAME[as.numeric(state.id)]
df <- cbind(df2, df1)
names(df)[1] <- "State"
```

Now the SpatialPointsDataFrame function can be used to convert the intersected spatial data (AoI.torn) into a format with attributes in the data frame, df, which can in turn be written to a shapefile for use in other applications:

```
AoI.torn <- SpatialPointsDataFrame(AoI.torn, data = df)
# write out as a shapefile if you wish
# writePointsShape(AoI.torn, "AoItorn.shp")
```

In the above example, the state names were attached to the output of the intersection. It is possible to extract and attach other attributes as well. The procedure below matches the state name from the intersection to the data held in us_states and then attaches this to the data frame of the intersection object, which can of course be converted to a SpatialPointsDataFrame variable. In effect the code attaches the data about the states to each tornado location:

```
# match df2 defined above to us_states$STATE_NAME
index2 <- match(df2, us_states$STATE_NAME)
# use this to select from the data frame of us_states
df3 <- data.frame(us_states)[index2,]
# bind together and rename the attribute
df3 <- cbind(df2, df1, df3)
names(df3)[1] <- "State"
# create spatial data
AoI.torn2 <- SpatialPointsDataFrame(AoI.torn, data = df3)
```

You should examine the help for gIntersection to see how it works and should note that it will operate on any pair of spatial objects provided they are projected using the same datum (in this case WGS84). In order to perform spatial operations you may need to re-project your data to the same datum using spTransform as described in Chapter 3.

5.3 BUFFERS

In many situations, we are interested in events or features that occur near to our area of interest as well as those within it. Environmental events such as tornados, for example, do not stop at state lines or other administrative boundaries. Similarly, if we were studying crime locations or spatial access to facilities such as shops or health services, we would want to know about locations near to the study area border. Buffer operations provide a convenient way of doing this, and buffers can be created in R using the gBuffer function.

Continuing with the example above, we might be interested in extracting the tornados occurring in Texas and those within 25 km of the state border. Thus the objective is to create a 25 km buffer around the state of Texas and to use that to select

from the tornado dataset. The gBuffer function in the rgeos package allows us to do that, and requires that a distance for the buffer is specified in terms of the units used in the projection. However, in order to do this, a different projection is required as distances are difficult to determine directly from projections in degrees (essentially the relationship between planar distance measures such as metres and kilometres to degrees varies with latitude). And gBuffer will return an error message if you try to buffer a non-projected spatial dataset. Therefore, the code below uses the projected US data, us_states2 and the resultant buffer is shown in Figure 5.3:

```
# select an Area of Interest and apply a buffer
AoI <- us_states2[us_states2$STATE_NAME == "Texas",]
AoI.buf <- gBuffer(AoI, width = 25000)
# map the buffer and the original area
par(mar=c(0,0,0,0))
plot(AoI.buf)
plot(AoI, add = T, border = "blue")
```

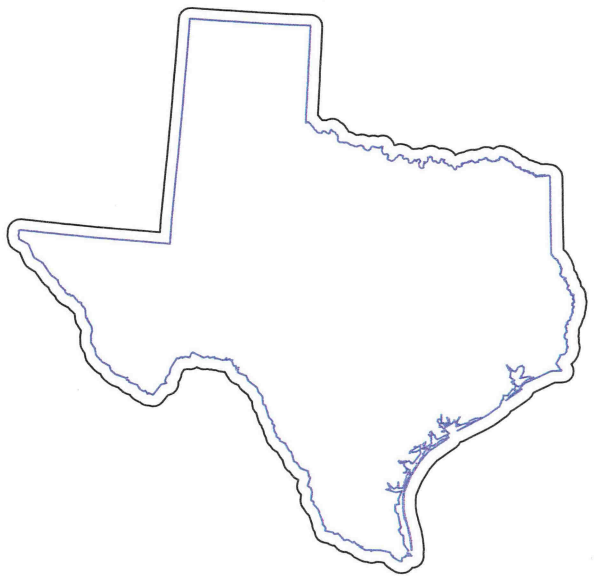

Figure 5.3 Texas with a 25 km buffer

The buffered object, shown in Figure 5.3 or objects can be used as input to gIntersection as above to expand the data that are extracted from the spatial overlay. You should also examine the impact on the output of other parameters in the gBuffer function that control how line segments are created, the geometry

of the buffer, join styles, etc. Also, you should note that any sp object can be used as an input to the gBuffer function: try applying it to the breach dataset that is put into working memory when the newhaven data are loaded.

There are number of options for defining how the buffer is created. If you enter the code below, using IDs, then buffers are created around each of the counties within the georgia2 dataset:

```
data(georgia)
# apply a buffer to each object
buf.t <- gBuffer(georgia2, width = 5000, byid = T,
    id = georgia2$Name)
# now plot the data
plot(buf.t)
plot(georgia2, add = T, border = "blue")
```

The IDs of the resulting buffer dataset relate to each of the input features, which in the above code has been specified to be the county names. This can be checked by examining how the buffer object has been named using names(buf.t). If you are not convinced that the indexing has been preserved then you can compare the output with a familiar subset, Appling County:

```
plot(buf.t[1,])
plot(georgia2[1,], add = T, col = "blue")
```

5.4 MERGING SPATIAL FEATURES

In the first intersection example above, four US states were selected and used to define the area of interest over which the tornado data were extracted. An attribute describing in which state each tornado occurred was added to the data frame of the intersected object. In other instances we may wish to consider the area as a single object and to merge the features within it. This can be done using the gUnaryUnion function in the rgeos package which was used in Chapter 3 to create an outline of the state of Georgia from its constituent counties. In the code below the US states are merged into a single object and the plotted over the original data as shown in Figure 5.4:

```
AoI.merge <- gUnaryUnion(us_states)
# now plot
par(mar=c(0,0,0,0))
plot(us_states, border = "darkgreen", lty = 3)
plot(AoI.merge, add = T, lwd = 1.5)
```

The gUnaryUnion function is one a set of union functions, the rest of which are described in the rgeos help section. It takes a variable of class SpatialPolygons

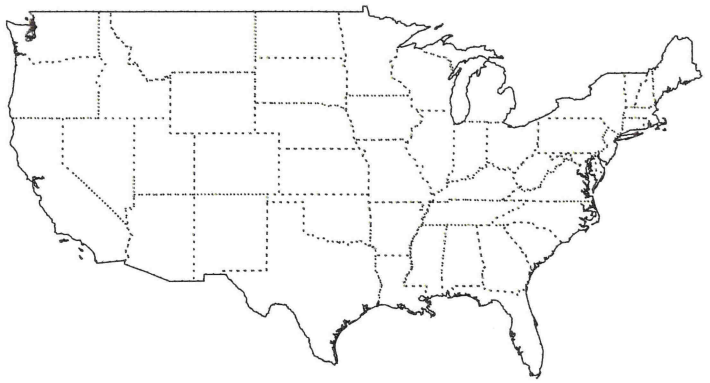

Figure 5.4 The outline of the merged US states created by gUnaryUnion, with the original state outlines in green

or SpatialPolygonsDataFrame with sub-geometries which it merges or, in set-theoretical terms, unions together. Once the merged objects have been created they can be used as inputs into the intersection and buffering procedures above in order to *select* data for analysis, as well as the analysis operations described below. The merged objects can also be used in a cartographic context to provide a border to the study area being considered.

5.5 POINT-IN-POLYGON AND AREA CALCULATIONS

5.5.1 Point-in-Polygon

It is often useful to count the number of points in a SpatialPoints dataset that fall inside each zone in a polygon dataset. This can be done using the poly.counts function in the GISTools package, which extends the gContains function in the rgeos package.

Remember that you can examine how a function works by entering it into the console without the brackets:

```
poly.counts

## function (pts, polys)
## colSums(gContains(polys, pts, byid = TRUE))
## <environment: namespace:GISTools>
```

The code below returns a list of counts of the number of tornados that occur inside each US state to the variable `torn.count` and prints the first six of these to the console using the `head` function:

```
torn.count <- poly.counts(torn, us_states)
head(torn.count)
```

```
##    1    2    3    4    5    6
##   79  341   87 1121 1445  549
```

The numbers along the top are the 'names' of the elements in the variable `tmp`, which in this case are the polygon ID numbers of the `us_states` variable. The values below are the counts of the points in the corresponding polygons. You can check this by entering:

```
names(torn.count)
```

5.5.2 Area Calculations

Another useful `GISTools` function is `poly.areas` which returns the area (in squared map units) of each polygon, using the `gArea` function in `rgeos`. To check the projection, and therefore the map units, of an `sp` class object (i.e. including `SpatialPolygons`, `SpatialPoints`, etc.), use the `proj4string` function:

```
proj4string(us_states2)
```

This declares the projection to be in metres. To see the areas in square metres of each US state, enter:

```
poly.areas(us_states2)
```

These are not particularly useful and more realistic measures are to report areas in hectares or square kilometres:

```
# hectares
poly.areas(us_states2) / (100 * 100)
# square kilometres
poly.areas(us_states2) / (1000 * 1000)
```

Self-Test Question 1. Your task is to create the code to produce maps of the densities of breaches of the peace in New Haven in *breaches per square mile*. For the analysis you will need to use the `breach` point data and the census `blocks` in the `newhaven` dataset using the `poly.counts` and `poly.areas` functions. The maps should be produced using the `choropleth` function. Remember that the New Haven data are included in the `GISTools` package:


```
library(GISTools)
data(newhaven)
```

As with all the self-test questions in this book, worked answers are provided at the end of the chapter.

You should note that the New Haven is projected in feet. Thus to report the breaches of the peace per square mile you will need to apply the `ft2miles` function to the results of the `poly.area` calculation, and as areas are in squared units, you will need to apply it twice:

```
ft2miles(ft2miles(poly.areas(blocks)))
```

5.5.3 Point and Areas Analysis Exercise

An important advantage of using R to handle spatial data is that it is very easy to incorporate your data into statistical analysis and graphics routines. For example, in the New Haven `blocks` data frame, there is a variable called `P_OWNEROCC` which states the percentage of owner-occupied housing in each census block. It may be of interest to see how this relates to the breach of the of peace densities calculated in Self-Test Question 1. A useful statistic is the correlation coefficient generated by the `cor` function which causes the correlation to be printed out:

```
data(newhaven)
densities= poly.counts(breach,blocks) /
    ft2miles(ft2miles(poly.areas(blocks)))
cor(blocks$P_OWNEROCC,densities)
```

```
## [1] -0.2038
```

In this case the two variables have a correlation of around –0.2, a weak negative relationship, suggesting that in general, places with a higher proportion of owner-occupied homes tend to see fewer breaches of peace. It is also possible to plot the relationship between the quantities – close the plot window if it is still open before running this code:

```
plot(blocks$P_OWNEROCC,densities)
```

A more detailed approach might be to model the number of breaches of peace. Typically, these are relatively rare, and a Poisson distribution might be an appropriate model. A possible model might then be:

```
breaches ~ Poisson(AREA * exp(a + b * blocks$P_OWNEROCC))
```

where AREA is the area of a block, P_OWNEROCC is the percentage of owner-occupiers in the block, and a and b are coefficients to be estimated, a being the intercept term. The AREA variable plays the role of an *offset* – a variable that always has a coefficient of 1. The idea here is that even if breaches of peace were uniformly distributed, the number of incidents in a given census block would be proportional to the AREA of that block. In fact, we can rewrite the model such that the offset term is the log of the area:

```
breaches ~ Poisson(exp(a + b * blocks$P_OWNEROCC+log(AREA)))
```

Seeing the model written this way makes it clear that the offset term has a coefficient that must always be equal to 1. The model can be fitted in R using the following code:

```
# load and attach the data
data(newhaven)
attach(data.frame(blocks))
# calculate the breaches of the peace in each block
n.breaches = poly.counts(breach,blocks)
area = ft2miles(ft2miles(poly.areas(blocks)))
# fit the model
model1=glm(n.breaches~P_OWNEROCC,offset=log(area),family=
    poisson)
# detach the data
detach(data.frame(blocks))
```

The first two lines compute the counts, storing them in n.breaches and the areas, storing them in area. The next line fits the Poisson model. glm stands for 'generalized linear model', and extends the standard lm routine to fit models such as Poisson regression. As a reminder, further information about linear models and the R modelling language was provided in one of the information boxes in Chapter 3 and an example of its use was given. The family=poisson option specifies that a Poisson model is to be fitted here. The offset option specifies the offset term, and the first argument specifies the actual model to be fitted. The model fitting results are stored in the variable model1. Having created the model in this way, entering

```
model1
```

returns a brief summary of the fitted model. In particular, it can be seen that the estimated coefficients are a = 3.02 and b = –0.0310. A more detailed view can be obtained using:

```
summary(model1)
```


Now, among other things, the standard errors and Wald statistics for a and b are shown. The Wald *Z* statistics are similar to *t statistics* in ordinary least squares regression, and may be tested against the normal distribution. The results below summarise the information, showing that both a and b are significant – and that therefore there is a statistically significant relationship between owner-occupation and breach of peace incidents:

	Estimate	Std. error	Wald's *Z*	*P*-value
Intercept	3.02	0.11	27.4	<0.01
Owner Occ. %	–0.031	0.00364	–8.5	<0.01

It also possible to extract diagnostic information from fitted models. For example, the rstandard function extracts the standardised residuals from a model. Whereas residuals are the difference between the observed value (i.e. in the data) and the value when estimated using the model, standardised residuals are rescaled to have a variance of 1. If the model being fitted is correct, then these residuals should be independent, have a mean of 0, a variance of 1 and an approximately normal distribution. One useful diagnostic is to map these values. The code below first computes them, and stores them in a variable called s.resids:

```
s.resids = rstandard(model1)
```

Now, to plot the map it will be more useful to specify a shading scheme directly using the shading command:

```
resid.shades = shading(c(-2,2),c("red","grey","blue"))
```

This specifies that the map will have three class intervals: below –2, between –2 and 2, and above 2. These are useful intervals given that the residuals should be normally distributed, and these values are the approximate two-tailed 5% points of this distribution. Residuals within these points will be shaded grey, large negative residuals will be red, and large positive ones will be blue:

```
par(mar=c(0,0,0,0))
choropleth(blocks,s.resids,resid.shades)
# reset the plot margins
par (mar=c(5,4,4,2))
```

From Figure 5.5 it can be seen that in fact there is notably more variation than one might expect (there are 21 blocks shaded blue or red, about 16% of the total, when around 5% would appear based on the model's assumptions), and also that the shaded blocks seem to cluster together. This last observation casts doubt on

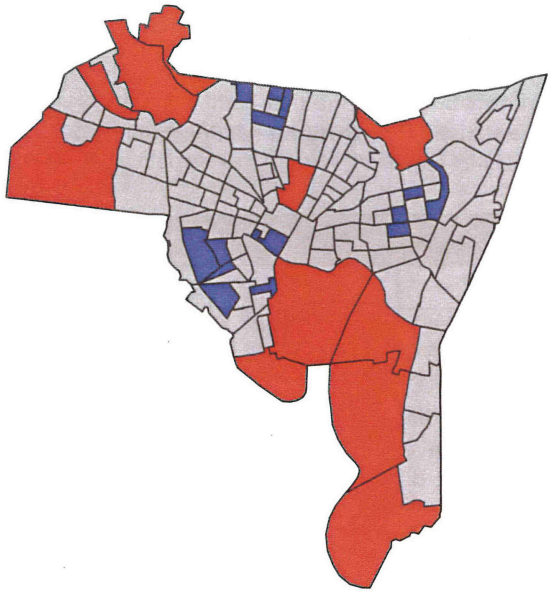

Figure 5.5 The distribution of the `model1` residuals, describing the relationship between breaches of the peace and owner-occupancy

the assumption of independence, suggesting instead that some degree of spatial correlation is present. One possible reason for this is that further variables may need to be added to the model, to explain this extra variability and spatial clustering amongst the residuals.

It is possible to extend this analysis by considering `P_VACANT`, the percentage of vacant properties in each census block, as well as `P_OWNEROCC`. This is done by extending `model1` and entering:

```
attach(data.frame(blocks))
n.breaches = poly.counts(breach,blocks)
area = ft2miles(ft2miles(poly.areas(blocks)))
model2=glm(n.breaches~P_OWNEROCC+P_VACANT,
    offset=log(area),family=poisson)
s.resids.2 = rstandard(model2)
detach(data.frame(blocks))
```

This sets up a new model, with a further term for the percentage of vacant housing in each block, and stores it in `model2`. Entering `summary(model2)` shows that the new predictor variable is significantly related to breaches of peace, with a

positive relationship. Finally, it is possible to map the standardised residuals for the new model reusing the shading scheme defined above:

```
s.resids.2 = rstandard(model2)
par(mar=c(0,0,0,0))
choropleth(blocks,s.resids.2,resid.shades)
# reset the plot margins
par (mar=c(5,4,4,2))
```

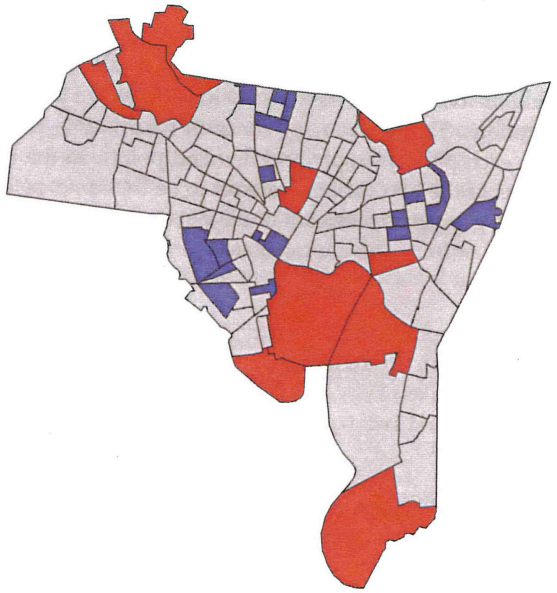

Figure 5.6 The distribution of the `model2` residuals, describing the relationship between breaches of the peace with owner-occupancy and vacant properties

This time, Figure 5.6 shows that there are fewer red and blue shaded census blocks – although perhaps still more than we might expect, and there is still some evidence of spatial clustering. Adding the extra variable has improved things to some extent, but perhaps there is more investigative research to be done. A more comprehensive treatment of spatial analysis of spatial data attributes is given in Chapter 7.

5.6 CREATING DISTANCE ATTRIBUTES

Distance is fundamental to spatial analysis. For example, we may wish to analyse the number of locations (health facilities, schools, etc.) within a certain distance of

the features we are considering. In the exercise below, distance measures are used to evaluate differences in accessibility for different social groups. These approaches form the basis of supply and demand modelling and provide inputs into location–allocation models.

Distance could be approximated using a series of buffers created at specific distance intervals around our features (whether point or polygons). These could be used to determine the number of features or locations that are within different distance ranges, as specified by the buffers using the `poly.counts` function above. However, the `gDistance` function calculates the Cartesian minimum (straight line) distance between two spatial datasets of class `sp`. In the code below, this function is used to determine the distances between the `places` variable (which are simply place holder locations for the names of districts New Haven but could be any kind of facility or *supply* feature), and the centroids of the census blocks in New Haven, in this case acting as *demand* locations. The `gDistance` function returns a to–from matrix of the distances between each pair of supply and demand points. In the first few lines of code, the projections of the two variables are set to be the same, before `gCentroid` is used to extract the geometric centroids of the census block areas and the distance between `places` and `centroids` are calculated:

```
data(newhaven)
proj4string(places) <- CRS(proj4string(blocks))
centroids. <- gCentroid(blocks, byid = T, id =
    rownames(blocks))
distances <- ft2miles(gDistance(places, centroids.,
    byid = T))
```

You can examine the result in relation to the inputs to `gDistance`, and you will see that the `distances` variable is a matrix of distances (in miles) from each of the 129 census block centroids to each of the nine locations described in the `places` variable. It is possible to use the census block polygons in the above `gDistance` calculation, but the distances returned will be to the nearest point of the census area. Using the census area centroid provides a more representative measure of the average distance experienced by people living in that area.

The `gWithinDistance` function tests whether each to–from distance pair is less than a specified threshold. It returns a matrix of `TRUE` and `FALSE` describing whether the distances between the elements of the two `sp` dataset elements are less than or equal to the specified distance or not. In the example below the distance specified is 1.2 miles.

```
distances <- gWithinDistance(places, blocks,
    byid = T, dist = miles2ft(1.2))
```

You should note that the `gDistance` and `gWithinDistance` functions work with whatever distance units are specified in the projections of the spatial features.

This means the inputs need to have the same units. Also remember that the `newhaven` data are projected in feet, hence the use of the `miles2ft` and `ft2miles` functions.

5.6.1 Distance Analysis/Accessibility Exercise

The use of distance measures in conjunction with census data is particularly useful for analysing access to the supply of some facility or service for different social groups. The code below replicates the analysis developed by Comber et al. (2008), examining access to green spaces. In this exercise a hypothetical example is used: we wish to examine the equity of access to the locations recorded in the `places` variable (supply) for different ethnic groups as recorded in the `blocks` dataset (demand), on the basis that we expect everyone to be within 1 mile of a facility. We will use the census data to approximate the number of people with and without access of less than 1 mile to the set of hypothetical facilities.

First, the distances variable is recalculated in case it was overwritten in the `gWithinDistance` example above. Then the minimum distance to a supply facility is determined for each census area using the `apply` function. Finally a logical statement is used to generate a `TRUE` or `FALSE` statement for each block:

```
distances <- ft2miles(gDistance(places, centroids., byid = T))
min.dist <- as.vector(apply(distances,1, min))
access <- min.dist < 1
# and this can be mapped
# plot(blocks, col = access)
```

The populations of each ethnic group in each census block can be extracted from the `blocks` dataset:

```
# extract the ethnicity data from the blocks variable
ethnicity <- as.matrix(data.frame(blocks[,14:18])/100)
ethnicity <- apply(ethnicity, 2, function(x) (x *
    blocks$POP1990))
ethnicity <- matrix(as.integer(ethnicity), ncol = 5)
colnames(ethnicity) <- c("White", "Black",
    "Native American", "Asian", "Other")
```

And then a crosstabulation is used to bring together the access data and the populations:

```
# use xtabs to generate a crosstabulation
mat.access.tab = xtabs(ethnicity~access)
# then transpose the data
```

```
data.set = as.data.frame(mat.access.tab)
#set the column names
colnames(data.set) = c("Access","Ethnicity", "Freq")
```

You should examine the `data.set` variable. This summarises all of the factors being considered: access, ethnicity and the counts associated with all factor combinations. If we make an assumption that there is an interaction between ethnicity and access, then this can be tested for using a generalised regression model with a Poisson distribution using the `glm` function:

```
modelethnic = glm(Freq~Access*Ethnicity,
    data=data.set,family=poisson)
# the full model can be printed to the console
# summary(modelethnic)
```

The model coefficient estimates show that there is significantly less access for some groups than would be expected under a model of equal access when compared to the largest ethnic group `White`, which was listed first in the `data.set` variable, and significantly greater access for the ethnic group `Other`. Examine the model coefficient estimates, paying particular attention to the `AccessTRUE:` coefficients:

```
summary(modelethnic)$coef
```

Then assign these to the a variable:

```
mod.coefs = summary(modelethnic)$coef
```

By subtracting 1 from the coefficients and converting them to percentages, it is possible to attach some likelihoods to the access for different groups when compared the ethnic group `White`. Again, you should examine the terms in the model outputs prefixed by `AccessTRUE:`, as below:

```
tab <- 100*(exp(mod.coefs[,1]) - 1)
tab <- tab[7:10]
names(tab) <- colnames(ethnicity)[2:5]
tab

##       Black Native American         Asian         Other
##      -35.08          -11.73        -29.83        256.26
```

The results in `tab` tell us that some ethnic groups have significantly less access to the hypothetical supply facilities when compared to the White ethnic group (as recorded in the census): the ethnic group Black have 35% less, Native Americans 12% less,

(although this is not significant), Asians 30% less and Other 256% more access than the White ethnic group.

It is possible to visualise the variations in access for different groups using a mosaic plot. Mosaic plots show the counts (i.e. population) as well as the residuals associated with the interaction between groups and their access, the full details of which were given in Chapter 3.

```
mosaicplot(t(mat.access.tab),xlab='',ylab='Access to Supply',
    main="Mosaic Plot of Access",shade=TRUE,las=3,cex=0.8)
```

Self-Test Question 2. In working through the exercise above you have developed a number of statistical techniques. In answering this self-test question you will explore the impact of using census data summarised over different areal units in your analysis. Specifically, you will develop and compare the results of two statistical models using different census areas in the `newhaven` datasets: `blocks` and `tracts`. You will analyse the relationship between residential property occupation and burglaries. You will need to work through the code below before the tasks associated with this questions are posited. To see the relationship between the census tracts and the census blocks, enter:

```
plot(blocks,border='red')
plot(tracts,lwd=2,add=TRUE)
```

You can see that the census blocks are nested within the tracts.

The analysis described below develops a statistical model to describe the relationship between residential property occupation and burglary using two of the New Haven crime variables related to residential burglaries. These are both point objects, called `burgres.f` and `burgres.n`. The first of these, `burgres.f`, is a list of burglaries where entry was forced into the property, and `burgres.n` is a list of burglaries where entry was not forced, suggesting that the property was left insecure, perhaps by leaving a door or window open. The burglaries data cover the six-month period between 1 August 2007 and 31 January 2008.

The questions you will consider are:

- Do both kinds of residential burglary occur in the same places – that is, if a place is a high-risk area for non-forced entry, does it imply that it is also a high-risk area for forced entry?
- How does this relationship vary over different census units?

To investigate these, you should use a bivariate regression model that attempts to predict the density of forced burglaries from the density of non-forced ones. The indicators needed for this are the rates of burglary given the number of properties

at risk. You should use the variable OCCUPIED, present in both the census blocks data frame and the the census tracts data frame, to estimate the number of properties at risk. If we were to compute rates per 1000 households, this would be: 1000*(number of burglaries in block)/OCCUPIED and since this is over a six-month period, doubling this quantity gives the number of burglaries per 1000 households per year. However, entering:

```
blocks$OCCUPIED
```

shows that some blocks have no occupied housing, so the above rate cannot be defined. To overcome this problem you should select the subset of the blocks with more than zero occupied dwellings. For polygon spatial objects, each individual polygon can be treated like a row in a data frame for the purposes of subset selection. Thus, to select only the blocks where the variable OCCUPIED is greater than zero, enter:

```
blocks2 = blocks[blocks$OCCUPIED > 0,]
```

We can now compute the burglary rates for forced and non-forced entries by first counting the burglaries in each block in blocks2 using the poly.counts function, dividing these numbers by the OCCUPIED counts and then multiplying by 2000 to get yearly rates per 1000 households. However, before we do this, you should remember that you need the OCCUPIED attribute from blocks2 and not blocks. Attach the blocks2 data and then calculate the two rate variables:

```
attach(data.frame(blocks2))
forced.rate = 2000*poly.counts(burgres.f,blocks2)/OCCUPIED
notforced.rate = 2000*poly.counts(burgres.n,blocks2)/
    OCCUPIED
detach(data.frame(blocks2))
```

You should have two rates stored in forced.rate and notforced.rate. A first attempt at modelling the relationship between the two rates could be via simple bivariate regression, ignoring any spatial dependencies in the error term. This is done using the lm function, which creates a simple regression model, model1:

```
model1 = lm(forced.rate~notforced.rate)
```

To examine the regression coefficients, enter:

```
summary(model1)
coef(model1)
```


The key things to note here are that the forced rate is related to the not-forced rate by the formula:

```
expected(forced rate) = a + b * (not forced rate)
```

where a is the intercept term and b is the slope or coefficient for the predictor variable. If the coefficient for the not-forced rate is statistically different from zero, indicated in the summary of the model, then there is evidence that the two rates are related. One possible explanation is that if burglars are active in an area, they will only use force to enter dwellings when it is necessary, making use of an insecure window or door if they spot the opportunity. Thus in areas where burglars are active, both kinds of burglary could potentially occur. However, in areas where they are less active it is less likely for either kind burglary to occur.

Having outlined the approach, your specific tasks in this question are:

1. To determine the coefficients a and b in the formula above for two different analyses using the blocks and tracts datasets.
2. To comment on the difference between the analyses using different areal units.

5.7 COMBINING SPATIAL DATASETS AND THEIR ATTRIBUTES

The point-in-polygon calculation using poly.counts generates counts of the points falling in each polygon. A common situation in spatial analysis is the need to combine (overlay) different polygon features that describe the spatial distribution of different variables, attributes or processes that are of interest. The problem is that the data may have different underlying area geographies. In fact, it is commonly the case that different agencies, institutions and government departments use different geographical areas, and even where they do not, geographical areas frequently change over time. In these situations, we can use the gIntersection function to identify the area of intersection between the datasets. With some manipulation it is possible to determine the proportions of the objects in dataset X that fall into each of polygon of dataset Y. This section uses a worked example to illustrate how this can be done in R.

In the subsequent self-test question you will develop a function to do this. As with all spatial operations on sp datasets, the input data need to have the same projections. You can examine their proj4string attributes to check and if need be use the spTransform function to put the data into the same projection.

A zone dataset will be created with the aim of calculating the number of houses in each zone. These will be extracted from the New Haven tracts data which includes the variable HSE_UNITS, describing the number of residential

properties in each census tract. The zones are hypothetical, but could perhaps be zones used by the emergency services for planning purposes and resource allocation.

First, you should create the zones, number them with an ID and plot these on a map with the tracts data. This is easily done by defining a grid and then converting this to a `SpatialPolygonsDataFrame` object. Enter:

```
data(newhaven)
# define sample grid in polygons
    bb <- bbox(tracts)
grd <- GridTopology(cellcentre.offset=
    c(bb[1,1]-200,bb[2,1]-200),
    cellsize=c(10000,10000), cells.dim = c(5,5))
int.layer <- SpatialPolygonsDataFrame(
    as.SpatialPolygons.GridTopology(grd),
    data = data.frame(c(1:25)), match.ID = FALSE)
names(int.layer) <- "ID"
```

Projections can be checked using `proj4string(int.layer)` and `proj4string(tracts)`. These have the same projections, in this case `NA`, and so they can be intersected:

```
int.res <- gIntersection(int.layer, tracts, byid = T)
```

You can examine the intersected data, the original data and the zones in the same plot window, as in Figure 5.7.

```
# set some plot parameters
par(mfrow = c(1,2))
par(mar=c(0,0,0,0))
# plot and label the zones
plot(int.layer, lty = 2)
Lat <- as.vector(coordinates(int.layer)[,2])
Lon <- as.vector(coordinates(int.layer)[,1])
Names <- as.character(data.frame(int.layer)[,1])
# plot the tracts
plot(tracts, add = T, border = "red", lwd =2)
pl <- pointLabel(Lon, Lat, Names, offset = 0, cex =.7)
# set the plot extent
plot(int.layer, border = "white")
# plot the intersection
plot(int.res, col=blues9, add = T)
```


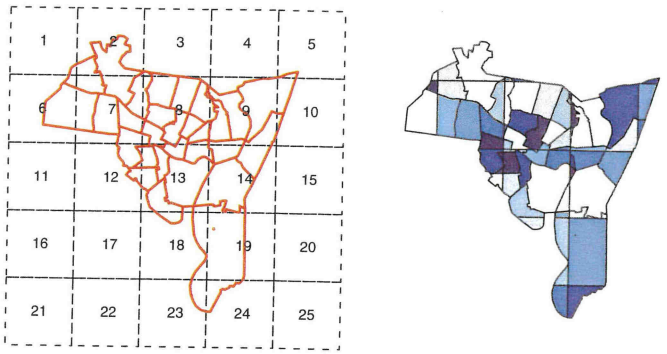

Figure 5.7 The zones and census tracts data before and after intersection

As in the `gIntersection` operation described in earlier sections, you can examine the result of the intersection:

```
names(int.res)
```

You will see that the names of the intersected objects are composites of the inputs. These can be used to link to the attributes held in the data frame of each input to the intersection, and then to create attributes for the intersection output data, in this case `int.res` and the original zone data `int.layer`.

First, the composite object names have to be split:

```
tmp <- strsplit(names(int.res), " ")
tracts.id <- (sapply(tmp, "[[", 2))
intlayer.id <- (sapply(tmp, "[[", 1))
```

Then, the proportions of the original `tract` areas need to be extracted – these will be used to proportionally allocate the counts of houses to the zones.

```
# generate area and proportions
int.areas <- gArea(int.res, byid = T)
tract.areas <- gArea(tracts, byid = T)
# match this to the new layer
index <- match(tracts.id, row.names(tracts))
tract.areas <- tract.areas[index]
tract.prop <- zapsmall(int.areas/tract.areas, 3)
# and create data frame for the new layer
df <- data.frame(intlayer.id, tract.prop)
houses <- zapsmall(tracts$HSE_UNITS[index] * tract.prop, 1)
df <- data.frame(df, houses, int.areas)
```

Finally, the attributes held in the new data frame, `df`, are summarised using `xtabs` and linked back to the original zone areas. Note that the `df` variable above could be attached to the `SpatialPolygonsDataFrame` object, `int.res`.

```
int.layer.houses <- xtabs(df$houses~df$intlayer.id)
index <- as.numeric(gsub("g", "", names(int.layer.houses)))
# create temporary variable
tmp <- vector("numeric", length = dim(data.frame(int.layer))[1])
tmp[index] <- int.layer.houses
i.houses <- tmp
```

Now the outputs can be attached to the original zone dataset:

```
int.layer <- SpatialPolygonsDataFrame(int.layer,
    data = data.frame(data.frame(int.layer),
    i.houses), match.ID = FALSE)
```

The results can be plotted as Figure 5.8 and checked against the original inputs in Figure 5.7:

```
# set the plot parameters and the shading variable
par(mar=c(0,0,0,0))
shades = auto.shading(int.layer$i.houses,
    n = 6, cols = brewer.pal(6, "Greens"))
# map the data
choropleth(int.layer, int.layer$i.houses, shades)
plot(tracts, add = T)
choro.legend(530000, 159115, bg = "white", shades,
    title = "No. of houses", under = "")
# reset the plot margins
par (mar=c(5,4,4,2))
```

Self-Test Question 3. Your task is to write a function that will return an intersected dataset, with an attribute of counts of some variable (houses, population, etc.) as held in another spatial polygon data frame. You should base your function on the code used in the illustrated example above. You should compile it such that the function returns the portion of the count attribute covered by each zone. For example, it should be able to intersect the `int.layer` data with the `blocks` data and return a `SpatialPolygonsDataFrame` dataset with an attribute of the number of people, as described in the `POP1990` variable of `blocks`, covered by each zone. You should remember that spatial functions such as `gIntersect` require their inputs to have the same projection. The `int.layer` defined above and the `tracts` data have no projections. You may find it useful to align the projections

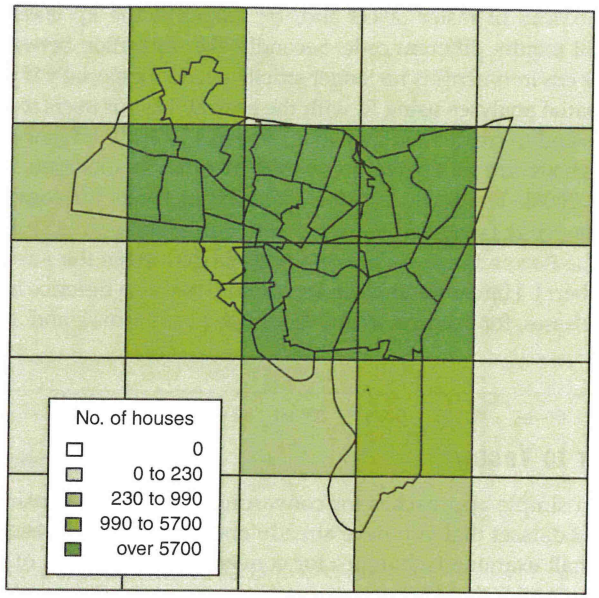

Figure 5.8 The zones shaded by the number of households after intersection with the census tracts

of the `int.layer` defined above and the `blocks` data in the following way using the `rgdal` package:

```
install.packages("rgdal", dep = T)
library(rgdal)
ct <- proj4string(blocks)
proj4string(int.layer) <- CRS(ct)
blocks <- spTransform(blocks, CRS(proj4string(int.layer)))
```

5.8 CONVERTING BETWEEN RASTER AND VECTOR

Very often we would like to move or convert our data between vector and raster environments. In fact the very persistence of these dichotomous data structures, with separate raster and vector functions and analyses in many commercial GIS software programs, is one of the long-standing legacies in GIS.

This section briefly describes methods for converting data between raster and vector structures. There are three reasons for this brief treatment. Firstly, many packages define their own data structures. For example, the functions in the `PBSmapping` package require a `PolySet` object to be passed to them. This means that conversion

between one class of raster object and, for example, the sp class of Spatial Polygons will require different code. Secondly, the separation between raster and vector analysis environments is no longer strictly needed, especially if you are developing your spatial analyses using R, with the easy ability for users to compile their own functions and to create their own analysis tools. Thirdly, advanced raster mapping and analysis are extensively covered in other books (see, for example, Bivand et al. 2008). The sections below describe methods for converting the sp class of objects (SpatialPoints, SpatialLines and SpatialPolygons, etc.) to and from the RasterLayer class of objects as defined in the raster package, created by Robert J. Hijmans and Jacob van Etten. They also describe how to convert between sp classes, for example to and from SpatialPixels and SpatialGrid objects.

5.8.1 Raster to Vector

In this section simple approaches for converting are illustrated using datasets in the tornados dataset that you have already encountered.

First, we shall examine techniques for converting the sp class of objects to the raster class, considering in turn:

- points (SpatialPoints and SpatialPointsDataFrame)
- lines (SpatialLines and SpatialLinesDataFrame)
- areas (SpatialPolygons and SpatialPolygonsDataFrame)

You will need to load the data and the packages – you may need to install the raster package using the install.packages function if this is the first time that you have used it.

Converting points to raster

```
library(GISTools)
library(raster)
data(tornados)
# Points
r = raster(nrow = 180, ncols = 360, ext = extent(us_states2))
t2 <- as(torn2, "SpatialPoints")
r <- rasterize(t2, r, fun=sum)
```

The resultant raster has cells describing different tornado densities that can be mapped as in Figure 5.9:


```
# set the plot extent by specifying the plot colour 'white'
plot(r, col = "white")
plot(us_states2, add = T, border = "grey")
plot(r, add = T)
```

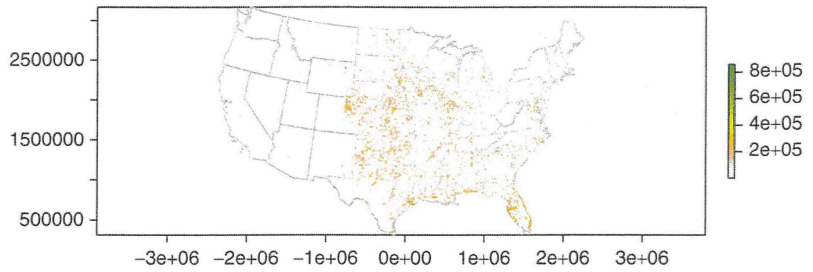

Figure 5.9 Converting points to raster format

Converting lines to raster

For illustrative purposes the code below creates a SpatialLinesDataFrame object of the outline of the polygons:

```
# Lines
us_outline <- as(us_states2 , "SpatialLinesDataFrame")
r <- raster(nrow = 180 , ncols = 360, ext = extent(us_states2))
r <- rasterize(us_outline , r, "STATE_FIPS")
```

This takes a bit longer to run, but again the results can be mapped and this time the shading describes the STATE_FIPS attribute – a numerical code for each US state (see Figure 5.10):

```
plot(r)
```

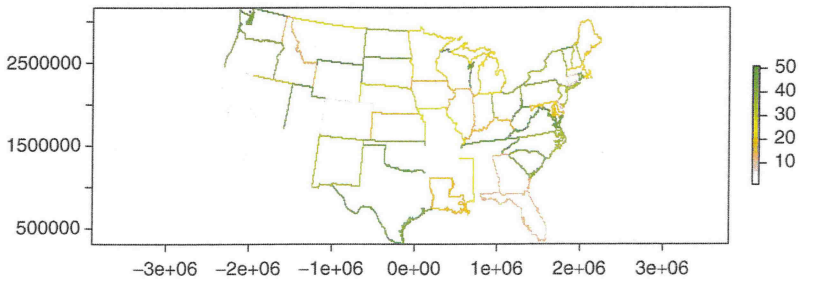

Figure 5.10 Converting lines to raster format

Converting polygons or areas to raster

Finally, polygons can easily be converted to a `RasterLayer` object using tools in the `raster` package and plotted as in Figure 5.11. You will note that in this case the 1997 population for each state is used to generate raster cell or pixel values.

```
# Polygons
r <- raster(nrow = 180 , ncols = 360, ext = extent(us_states2))
r <- rasterize(us_states2, r, "POP1997")

## Found 49 region(s) and 95 polygon(s)

plot(r)
```

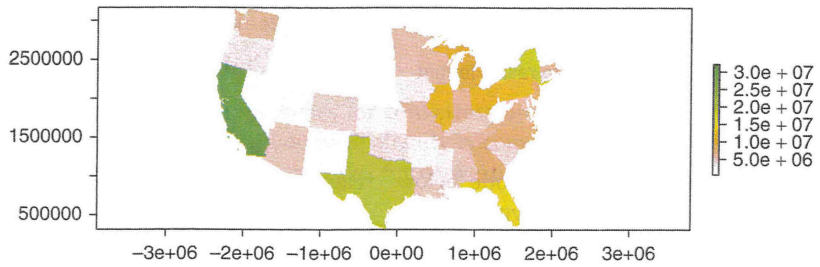

Figure 5.11 Converting polygons to raster format

It is instructive to examine the outputs of these processes. Enter:

```
r
```

This summarises the characteristics of the `raster` object, including the resolution, dimensions and extent.

It is possible to specify particular dimensions for the raster grid cells, rather than just dividing the dataset's extent by `ncol` and `nrow` in the `raster` function. The code below is a bit convoluted but cleanly allocates the values to raster grid cells of a specified size:

```
# specify a cell size in the projection units
d <- 50000
dim.x <- d
dim.y <- d
```



```
bb <- bbox(us_states2)
# work out the number of cells needed
cells.x <- (bb[1,2]-bb[1,1]) / dim.x
cells.y <- (bb[2,2]-bb[2,1]) / dim.y
round.vals <- function(x){
    if(as.integer(x) < x) {
       x <- as.integer(x) + 1
    } else {x <- as.integer(x)
            }}
# the cells cover the data completely
cells.x <- round.vals(cells.x)
cells.y <- round.vals(cells.y)
# specify the raster extent
ext <- extent(c(bb[1,1], bb[1,1]+(cells.x*d),
    bb[2,1],bb[2,1]+(cells.y*d)))
# now run the raster conversion
r <- raster(ncol = cells.x,nrow =cells.y)
extent(r) <- ext
r <- rasterize(us_states2, r, "POP1997")
# and examine the results
r
plot(r)
```

5.8.2 Converting to `sp` Classes

You may have noticed that the `sp` package also has two data classes that are able to represent raster data, or data are located on a regular grid. These are `SpatialPixelsDataFrame` and `SpatialGridDataFrame`. It is possible to convert the `raster` class objects using the `as` function. The example below converts the `raster` layer to `SpatialPixelsDataFrame` and to `SpatialGridDataFrame` objects.

First create a spatially coarse raster layer of US states similar to the above:

```
r <- raster(nrow = 60, ncols = 120, ext = extent(us_states2))
r <- rasterize(us_states2, r, "STATE_FIPS")

## Found 49 region(s) and 95 polygon(s)
```

Then the `as` function can be used to coerce this to `SpatialPixelsDataFrame` and `SpatialGridDataFrame` objects, which can also be mapped using the `image` or `plot` commands in the usual way, as in Figure 5.12:

```
g <- as(r, 'SpatialGridDataFrame')
p <- as(r, 'SpatialPixelsDataFrame')
# not run
# image(g, col = topo.colors(51))
# points(p, cex = 0.5)
par(mar=c(0,0,0,0))
plot(p, cex = 0.5, pch = 1, col = p$layer)
```

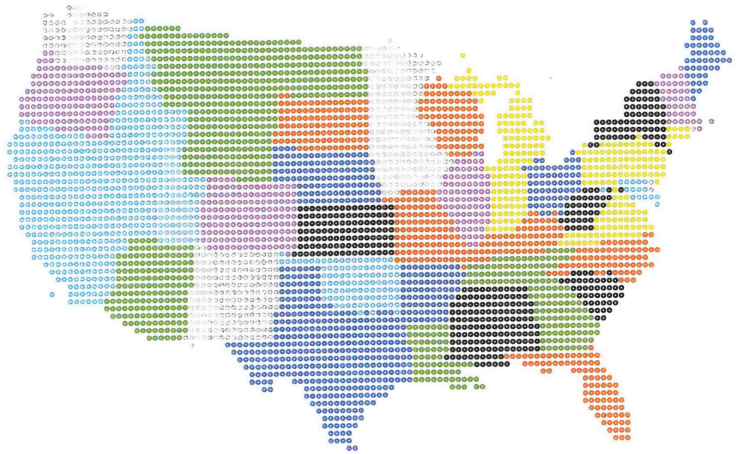

Figure 5.12 Plotting the `SpatialGrid` and `SpatialPoint` objects

You can also examine the data values held in the `data.frame` by entering:

```
head(data.frame(g))
head(data.frame(p))
```

The data can also be manipulated to select certain features, in this case selecting the states with populations greater than 10 million people. The code below assigns `NA` values to the data points that fail this test and plots the data as an `image` and as points (Figure 5.13):

```
# set up and create the raster
r <- raster(nrow = 60 , ncols = 120, ext = extent(us_states2))
r <- rasterize(us_states2 , r, "POP1997")

## Found 49 region(s) and 95 polygon(s)

r2 <- r
```



```
# subset the data
r2[r < 10000000] <- NA
g <- as(r2, 'SpatialGridDataFrame')
p <- as(r2, 'SpatialPixelsDataFrame')
# not run
# image(g, bg = "grey90")
par(mar=c(0,0,0,0))
plot(p, cex = 0.5, pch = 1)
```

Figure 5.13 Selecting data in `SpatialGrid` and `SpatialPoint` objects

5.8.3 Vector to Raster

The `raster` package contains a number of functions for converting from vector to raster formats. These include `rasterToPolygons` which converts to a `SpatialPolygonsDataFrame` object, and `rasterToPoints` which converts to a `matrix` object. Both are illustrated in the code below and the results shown in Figure 5.14. Notice how the original raster imposes a grid structure on the polygons that are created.

```
# load the data and convert to raster
data(newhaven)
# set up the raster, r
r <- raster(nrow = 60 , ncols = 60, ext = extent(tracts))
# convert polygons to raster
r <- rasterize(tracts , r, "VACANT")

## Found 29 region(s) and 30 polygon(s)

poly1 <- rasterToPolygons(r, dissolve = T)
```

```
# convert to points
points1 <- rasterToPoints(r)
# plot the points, rasterized polygons & orginal polygons
par(mar=c(0,0,0,0))
plot(points1, col = "grey", axes = FALSE, xaxt='n',
    ann=FALSE)
plot(poly1, lwd = 1.5, add = T)
plot(tracts, border = "red", add = T)
# reset the plot margins
par(mar=c(5,4,4,2))
```

Figure 5.14 Converting from rasters to polygons and points, with the original polygon data in red

5.9 INTRODUCTION TO RASTER ANALYSIS

This section provides the briefest of overviews of how raster data may be manipulated and overlayed in a R in a similar way to a standard GUI GIS such as QGIS. This section will introduce the `raster` package, the reclassification of raster data as a precursor to some basic methods for performing what is sometimes referred to as *map algebra*, using a *raster calculator* or *raster overlay*. As a

reminder, many packages include user guides in the form of a PDF document describing the package. This is listed at the top of the package index page. The `raster` package includes example code for the creation of raster data and different types of multi-layered raster composites. These will not be covered in this section. Rather, the coded examples illustrate some basic methods for manipulating and analysing raster layers in a similar way to what is often referred to as 'multi-criteria evaluation' or 'multi-criteria analysis'.

Raster analysis requires that the different data have a number of characteristics in common: typically they should cover the same spatial extent, the same spatial resolution (grid or cell size), and as with data for any spatial analysis, they should have the same projection or coordinate system. The data layers used in the example code in this section all have these properties. When you come to develop your own analyses, you may have to perform some manipulation of the data prior to analysis to ensure that your data also have these properties.

5.9.1 Raster Data Preparation

The Meuse data in the `sp` package will be used to illustrate the functions below. You could read in your raster data using the `readGDAL` function in the `rgdal` package, which provides an excellent engine for reading most commonly used raster formats. You can inspect the properties and attributes of the Meuse data by examining the associated help files `?meuse.grid`.

```
library(GISTools)
library(raster)
library(sp)
# load the meuse.grid data
data(meuse.grid)
# create a SpatialPixelsDataFrame object
coordinates(meuse.grid) <- ~x+y
meuse.grid <- as(meuse.grid, "SpatialPixelsDataFrame")
# create 3 raster layers
r1 <- raster(meuse.grid, layer = 3) #dist
r2 <- raster(meuse.grid, layer = 4) #soil
r3 <- raster(meuse.grid, layer = 5) #ffreq
```

The code above loads the `meuse.grid` data, converts it to a `SpatialPixelsDataFrame` format and then creates three separate raster layers in the `raster` format. These three layers will form the basis of the analyses in this section. You could visually inspect their attributes by using some simple `image` commands:

```
image(r1, asp = 1)
image(r2, asp = 1)
image(r3, asp = 1)
```

5.9.2 Raster Reclassification

Raster analyses frequently employ simple numerical and mathematical operations. In essence they allow you to add, multiply, subtract, etc. raster data layers, and these operations are performed on a cell by cell basis. So for an addition this might be in the form:

```
Raster_Result <- Raster.Layer.1 + Raster.Layer.2
```

Remembering that raster data are numerical, if the `Raster.Layer.1` and `Raster.Layer.2` data both contained the values 1, 2 and 3, it would be difficult to know the origin, for example, of a value of 3 in the `Raster_Result` output. The `r2` and `r3` layers created above both contain values in the range 1–3 describing soil types and flooding frequency, respectively (as described in the help for the `meuse.grid` data). Therefore we may wish to reclassify them in some way to understand the results of any overlay operation.

It is possible to reclassify raster data in a number of ways.

First, the raster data values can be manipulated using simple mathematical operations. These produce raster outputs describing the mathematical combination of the input raster layers. The code below multiplies one of the layers by 10. This means that the result combining both raster data layers using the add (+) function contains a fixed set of values – in this case 9 – which are tractable to the combinations of inputs used. A value of 32 would indicate values of 3 in `r3` (a flooding frequency of one in 50 years) and 2 in `r2` (a soil type of 'Rd90C/VII', whatever that is). The results of this simple overlay are shown in Figure 5.15 and in the table of values printed. Note the use of the `spplot` function in the code below.

```
Raster_Result <- r2 + (r3 * 10)

table(as.vector(Raster_Result$values))

##

##  11  12 13  21  22  23  31  32  33

## 535 242  2 736 450 149 394 392 203

spplot(Raster_Result, col.regions=brewer.pal(9, "Spectral"),
    cuts=8)
```

A second approach to reclassifying raster data is to employ logical operations on the data layers prior to combining them. These return `TRUE` or `FALSE` for each raster grid cell, depending on whether it satisfies the logical condition. The resultant layers can then be combined in mathematical operations as above. For example,

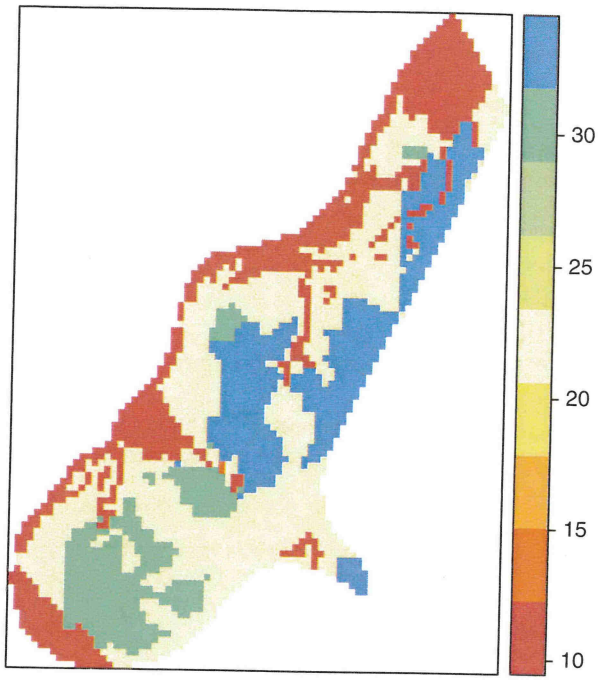

Figure 5.15 The result of a simple raster overlay

consider the analysis that wanted to identify the locations in the Meuse data that satisfied the following conditions:

- Are greater than half of the rescaled distance away from the Meuse river;
- Have a soil class of 1, i.e. calcareous weakly developed meadow soils, light sandy clay;
- Have a flooding frequency class of 3, i.e. once in a 50-year period.

The following logical operations can be used to do this:

```
r1a <- r1 > 0.5
r2a <- r2 >= 2
r3a <- r3 < 3
```

These can then be combined using specific mathematical operations, depending on the analysis. For example, a simple suitability Multi-Criteria Evaluation, where all

the conditions have to be true and where a crisp, Boolean output is required, would be coded using the multiplication function as below with the result shown in Figure 5.16:

```
Raster_Result <- r1a * r2a * r3a
table(as.vector(Raster_Result$values))

##
##    0    1
## 2924  179

plot(Raster_Result, legend = F, asp = 1)
# add a legend
legend(x='bottomright', legend = c("Suitable", "Not
    Suitable"), fill = (terrain.colors(n = 2)), bty = "n")
```

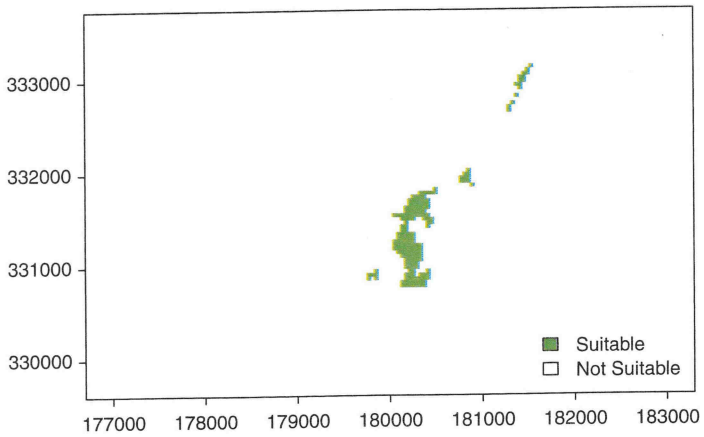

Figure 5.16 A raster overlay using a combinatorial AND

This is equivalent to a combinatorial AND operation, also known as an intersection. Alternatively the analysis may be interested in identifying where any of the conditions are true, a combinatorial OR operation, also known as a union, with a different result as shown in Figure 5.17:

```
Raster_Result <- r1a + r2a + r3a
table(as.vector(Raster_Result$values))
```



```
##
##    0    1    2    3
##  386 1526 1012  179

# plot the result and add a legend
image(Raster_Result, col = heat.colors(3), asp = 1)
legend(x='bottomright',
    legend = c("1 Condition", "2 Conditions",
    "3 Conditions"),
    fill = (heat.colors(n = 3)), bty = "n")
```

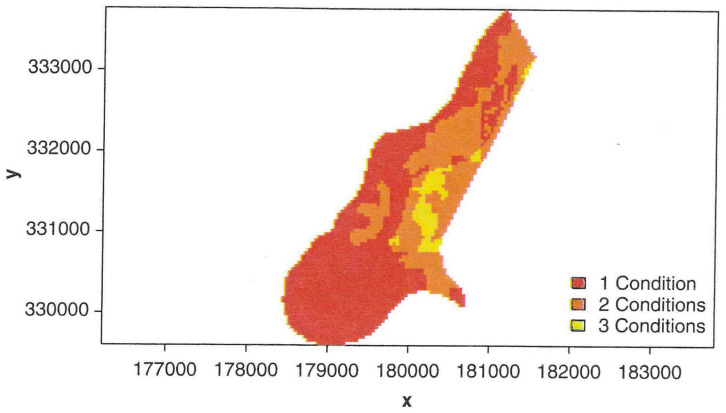

Figure 5.17 A raster overlay using a combinatorial OR

5.9.3 Other Raster Calculations

The above examples illustrated code to reclassify raster layers and then combined them using simple mathematical operations. You should note that it is possible to apply any kind of mathematical function to a raster layer. For example:

```
Raster_Result <- sin(r3) + sqrt(r1)
Raster_Result <- ((r1 * 1000 ) / log(r3) ) * r2
image(Raster_Result)
```

A number of other operations are possible using different functions included in the `raster` package. They are not given a full treatment here but are introduced such that the interested reader can explore them in more detail.

The `calc` function performs a computation over a single raster layer, in a similar manner to the mathematical operations in the preceding text. The advantage of

the `calc` function is that it should be faster when computing more complex operations over large raster datasets.

```
my.func <- function(x) {log(x)}
Raster_Result <- calc(r3, my.func)
# this is equivalent to
Raster_Result <- calc(r3, log)
```

The `overlay` function provides an alternative to the mathematical operations illustrated in the reclassification examples above for combining multiple raster layers. The advantage of the `overlay` function, again, is that it is more efficient for performing computations over large `raster` objects.

```
Raster_Result <- overlay(r2,r3,
    fun = function(x, y) {return(x + (y * 10))} )
# alternatively using a stack
my.stack <- stack(r2, r3)
Raster_Result <- overlay(my.stack, fun = function(x, y)
    (x + (y * 10)) )
```

There are a number of distance functions for computing distances to specific features. The `distanceFromPoints` calculates the distance between a set of points to all cells in a raster surface and produces a distance or cost surface as in Figure 5.18.

```
# load meuse and convert to points
data(meuse)
coordinates(meuse) <- ~x+y
# select a point layer
soil.1 <- meuse[meuse$soil == 1,]
# create an empty raster layer
# this is based on the extent of meuse
r <- raster(meuse.grid)
dist <- distanceFromPoints(r, soil.1)
plot(dist)
plot(soil.1, add = T)
```

You are encouraged to explore the `raster` package (and indeed the `sp` package) in more detail if you are specifically interested in raster based analyses. There are a number of other distance functions, functions for computing over neighbourhoods (focal functions), accessing raster cell values and assessing spatial configurations of raster layers.

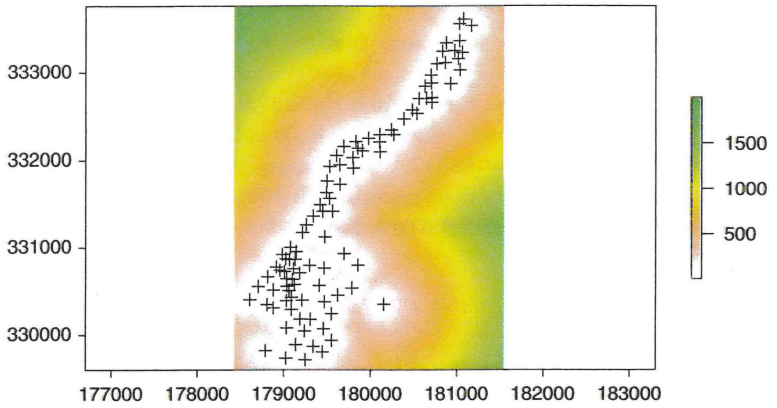

Figure 5.18 A raster analysis of distance to points

ANSWERS TO SELF-TEST QUESTIONS

Q1. Here is the code to maps the densities of breaches of the peace in New Haven in breaches per square mile:

```
densities = poly.counts(breach,blocks) /
    ft2miles(ft2miles(poly.areas(blocks)))
density.shades <- auto.shading(densities,
    cols=brewer.pal(5, "Oranges"), cutter=rangeCuts)
choropleth(blocks,densities,shading=density.shades)
choro.legend(533000,161000,density.shades)
title("Incidents per Sq. Mile")
```

Note that much of the concentration of incidents occurs in a small number of places.

Q2. First, calculate the coefficients for the analysis using census `blocks`:

```
# Analysis with blocks
blocks2 = blocks[blocks$OCCUPIED > 0,]
attach(data.frame(blocks2))
forced.rate = 2000*poly.counts(burgres.f,blocks2)/OCCUPIED
notforced.rate = 2000*poly.counts(burgres.n,blocks2)/
    OCCUPIED
model1 = lm(forced.rate~notforced.rate)
coef(model1)

##    (Intercept) notforced.rate
##          5.467          0.379
```

```
cat("expected(forced rate)= ",coef(model1)[1], "+",
    coef(model1)[2], "* (not forced rate) ")

## expected(forced rate)= 5.467 + 0.379 * (not forced rate)

detach(data.frame(blocks2))
```

Now, calculate the coeffcients using census `tracts`:

```
# Analysis with tracts
tracts2 = tracts[tracts$OCCUPIED > 0,]
# align the projections
ct <- proj4string(burgres.f)
proj4string(tracts2) <- CRS(ct)
# now do the analysis
attach(data.frame(tracts2))
forced.rate = 2000*poly.counts(burgres.f,tracts2)/OCCUPIED
notforced.rate = 2000*poly.counts(burgres.n,tracts2)/OCCUPIED
model2 = lm(forced.rate~notforced.rate)
coef(model2)

##  (Intercept) notforced.rate
##       5.2435         0.4133

cat("expected(forced rate) = ",coef(model2)[1], "+",
    coef(model2)[2], "* (not forced rate) ")
## expected(forced rate) =  5.243 + 0.4133 * (not forced
rate)

detach(data.frame(tracts2))
```

These two analyses show that, in this case, there are only small differences between the coefficients arising from analyses using different areal units.

```
cat("expected(forced rate) = ",coef(model1)[1], "+",
    coef(model1)[2], "* (not forced rate) ")

## expected(forced rate) = 5.467 + 0.379 * (not forced rate)

cat("expected(forced rate) = ",coef(model2)[1], "+",
    coef(model2)[2], "* (not forced rate)")

## expected(forced rate) = 5.243 + 0.4133 * (not forced rate)
```


This analysis tests what is referred to as the modifiable areal unit problem, first identified in the 1930s, and extensively researched by Stan Openshaw in the 1970s and beyond – see Openshaw (1984) for a comprehensive review. Variability in analyses can arise when data are summarised over different spatial units, and the importance of the modifiable areal unit problem cannot be overstated as a critical consideration in spatial analysis.

Q3. The simplest way to write the function required would be simply to use the code in the preceding text and to wrap it in a function:

```
int.poly.counts <- function(int.layer, tracts,
    tracts.var, var.name) {
  int.res <- gIntersection(int.layer, tracts, byid = T)
  # split the intersection references
  tmp <- strsplit(names(int.res), " ")
  tracts.id <- (sapply(tmp, "[[", 2))
  intlayer.id <- (sapply(tmp, "[[", 1))
  # calculate areas
  int.areas <- gArea(int.res, byid = T)
  tract.areas <- gArea(tracts, byid = T)
  # match this to the new layer
  index <- match(tracts.id, row.names(tracts))
  tract.areas <- tract.areas[index]
  tract.prop <- zapsmall(int.areas/tract.areas, 3)
  # and create data frame for the new layer
  df <- data.frame(intlayer.id, tract.prop)
  houses <- zapsmall(tracts.var[index] * tract.prop, 1)
  df <- data.frame(df, houses, int.areas)
  # Finally, link back to the original areas
  int.layer.houses <- xtabs(df$houses~df$intlayer.id)
  index <- as.numeric(gsub("g", "", names(int.layer.houses)))
  # create temporary variable
  tmp <- vector("numeric", length = dim(data.frame(int.layer))[1])
  tmp[index] <- int.layer.houses
  i.houses <- tmp
  # create output data
  int.layer2 <- SpatialPolygonsDataFrame(int.layer,
    data = data.frame(data.frame(int.layer), i.houses),
    match.ID = FALSE)
  names(int.layer2) <- c("ID", var.name)
  return(int.layer2)
}
```

And this could be used to evaluate the inputs as in the worked example:

```
int.layer2 <- int.poly.counts(int.layer,
    tracts,tracts$HSE_UNITS, "i.house" )
```

However, the code is not very transparent and better names could be used for the various intermediate internal variables that are created to make the function more understandable to someone else or you at a later date.

```
int.poly.counts <- function(int.layer1, int.layer2,
    int.layer2.var, var.name) {
  int.res <- gIntersection(int.layer1, int.layer2, byid = T)
  tmp <- strsplit(names(int.res), " ")
  int.layer2.id <- (sapply(tmp, "[[", 2))
  intlayer.id <- (sapply(tmp, "[[", 1))
  int.areas <- gArea(int.res, byid = T)
  tract.areas <- gArea(int.layer2, byid = T)
  index <- match(int.layer2.id, row.names(int.layer2))
  tract.areas <- tract.areas[index]
  tract.prop <- zapsmall(int.areas/tract.areas, 3)
  df <- data.frame(intlayer.id, tract.prop)
  var <- zapsmall(int.layer2.var[index] * tract.prop, 1)
  df <- data.frame(df, var, int.areas)
  int.layer1.var <- xtabs(df$var~df$intlayer.id)
  index <- as.numeric(gsub("g", "", names(int.layer1.var)))
  tmp<-vector("numeric", length=dim(data.frame(int.layer1))[1])
  tmp[index] <- int.layer1.var
  i.var <- tmp
  int.layer.out <- SpatialPolygonsDataFrame(int.layer1,
    data = data.frame(data.frame(int.layer1), i.var),
    match.ID = FALSE)
  names(int.layer.out) <- c("ID", var.name)
  return(int.layer.out)}
```

This can then be tentatively applied to other data, after making sure that it has a similar projected (i.e. consistent with distance and area calculations) coordinate system. A full implementation of this function and the results of applying it to `int.layer` and `blocks`, after they have had their spatial reference systems aligned, is described below and shown in Figure 5.19.

```
# Set up the packages and data
```



```
library(GISTools)
library(rgdal)
data(newhaven)
# define the intersection layer just to make sure
bb <- bbox(tracts)
grd <- GridTopology(cellcentre.offset=
    c(bb[1,1]-200,bb[2,1]-200),
    cellsize=c(10000,10000), cells.dim = c(5,5))
int.layer <- SpatialPolygonsDataFrame(
    as.SpatialPolygons.GridTopology(grd),
    data = data.frame(c(1:25)), match.ID = FALSE)
names(int.layer) <- "ID"

# now run with some data
# match prj4strings
ct <- proj4string(blocks)
proj4string(int.layer) <- CRS(ct)
int.layer <- spTransform(int.layer,
    CRS(proj4string(blocks)))
# now run the function
int.result <- int.poly.counts(int.layer, blocks,
    blocks$POP1990, "i.pop" )
# set plot parameters
par(mar=c(0,0,0,0))

# map the results
shades = auto.shading(int.result$i.pop,n = 5,
    cols = brewer.pal(5, "OrRd"))
choropleth(int.result, int.result$i.pop, shades)
plot(blocks, add = T, lty = 2, lwd = 1.5)
choro.legend(530000, 159115, bg = "white", shades,
    title = "Count", under = " ")
```

You can check the assigned populations in relation to Figures 5.7 and 5.19.

```
matrix(data.frame(int.result)[,2], nrow = 5, ncol = 5,
    byrow = T)
```

```
##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,]  154  5682   556     0   236
## [2,] 1962 20354 41712 17125  3088
## [3,]    0  3476 20603 10494     0
## [4,]    0     0   587  4054     0
## [5,]    0     0   208   261     0
```

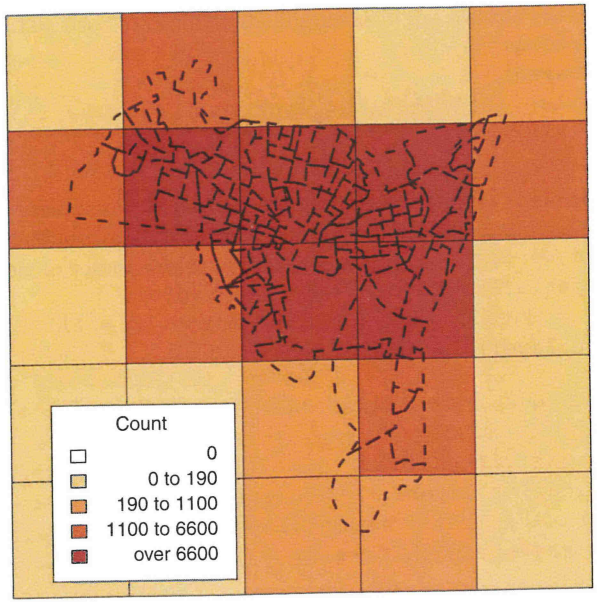

Figure 5.19 The zones shaded by population after intersection with the census blocks

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.