R Companion to O'Sullivan and Unwin

Robert J. Hijmans

Nov 30, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

These pages accompany the book "Geographic Information Analysis" by David O'Sullivan and David J. Unwin (2nd Edition, 2010) – hereinafter referred to as "OSU".

OSU is an excellent and very accessible introduction to spatial data analysis; but it does not show how to practically implement the methods that are discussed.

Many of the numerical examples in the text are implemented here, and some of the other techniques discussed are illustrated as well. We hope that this allows readers of OSU to get a more hands-on way to understand the material covered; and to apply such approaches in their own work. Throughout these pages reference is made to OSU, and no attempt is made to explain the material to those who have no access to OSU.

The examples are all implemented with *R*. If you are new to *R*, first go through this introduction, and make sure to read a bit about spatial data handling in R as well.

CHAPTER

TWO

THE LENGTH OF A COASTLINE

This page accompanies Chapter 1 of O'Sullivan and Unwin (2010). There is only one numerical example in this chapter, and it is a complicated one. I reproduce it here anyway, perhaps you can revisit it when you reach the end of the book (and you will be amazed to see how much you have learned!).

On page 13 the fractional dimension of a part of the New Zealand coastline is computed. First we get a high spatial resolution (30 m) coastline.

Throughout this book, we will use data that is installed with the `rspat` package. To install this package (from github) you can use the `install_github` function from the `remotes` package (so you may need to run `install.packages("remotes")` first.

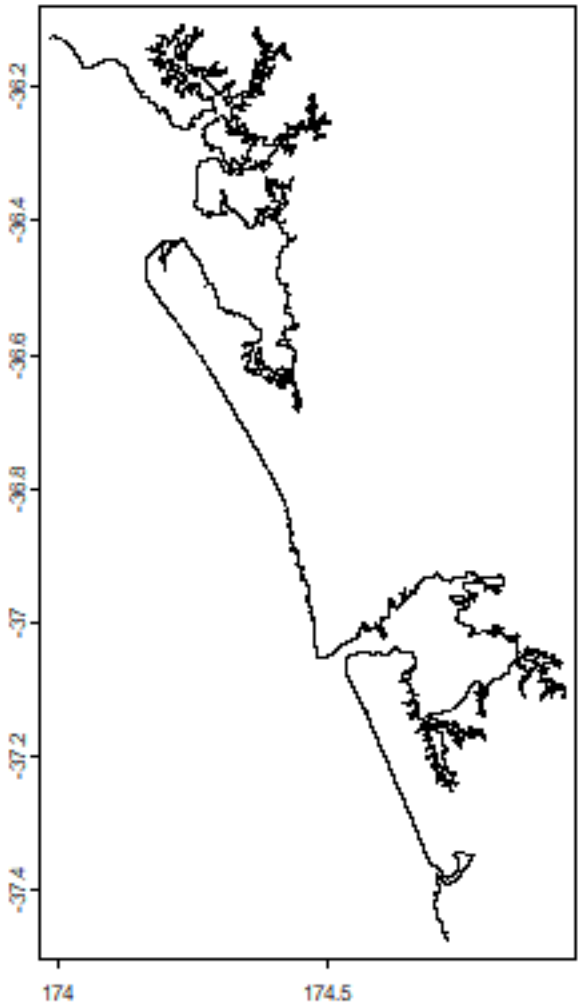
```
if (!require("remotes")) install.packages("remotes")
## Loading required package: remotes
```

Now install `rspat`

```
if (!require("rspat")) remotes::install_github("rspatial/rspat")
## Loading required package: rspat
## Loading required package: terra
## terra 1.7.62
```

Now you should have the data for all chapters.

```
library(terra)
library(rspat)
coast <- spat_data("nz_coastline")
coast
##  class       : SpatVector
##  geometry    : lines
##  dimensions  : 1, 0  (geometries, attributes)
##  extent      : 173.9854, 174.9457, -37.47378, -36.10576  (xmin, xmax, ymin, ymax)
##  coord. ref. : +proj=longlat +ellps=WGS84 +towgs84=0,0,0,0,0,0,0 +no_defs
plot(coast)
```



To speed up the distance computations, we transform the CRS from longitude/latitude to a planar system.

```
prj <- "+proj=tmerc +lat_0=0 +lon_0=173 +k=0.9996 +x_0=1600000 +y_0=10000000␣
↪+datum=WGS84 +units=m"
mcoast <- project(coast, prj)
mcoast
##  class       : SpatVector
##  geometry    : lines
##  dimensions  : 1, 0  (geometries, attributes)
##  extent      : 1688669, 1772919, 5851165, 6003617  (xmin, xmax, ymin, ymax)
##  coord. ref. : +proj=tmerc +lat_0=0 +lon_0=173 +k=0.9996 +x_0=1600000 +y_0=10000000␣
↪+datum=WGS84 +units=m +no_defs
```

On to the tricky part. A function to follow the coast with a yardstick of a certain length.

Argument `x` is a matrix with two columns (x and y coordinates)

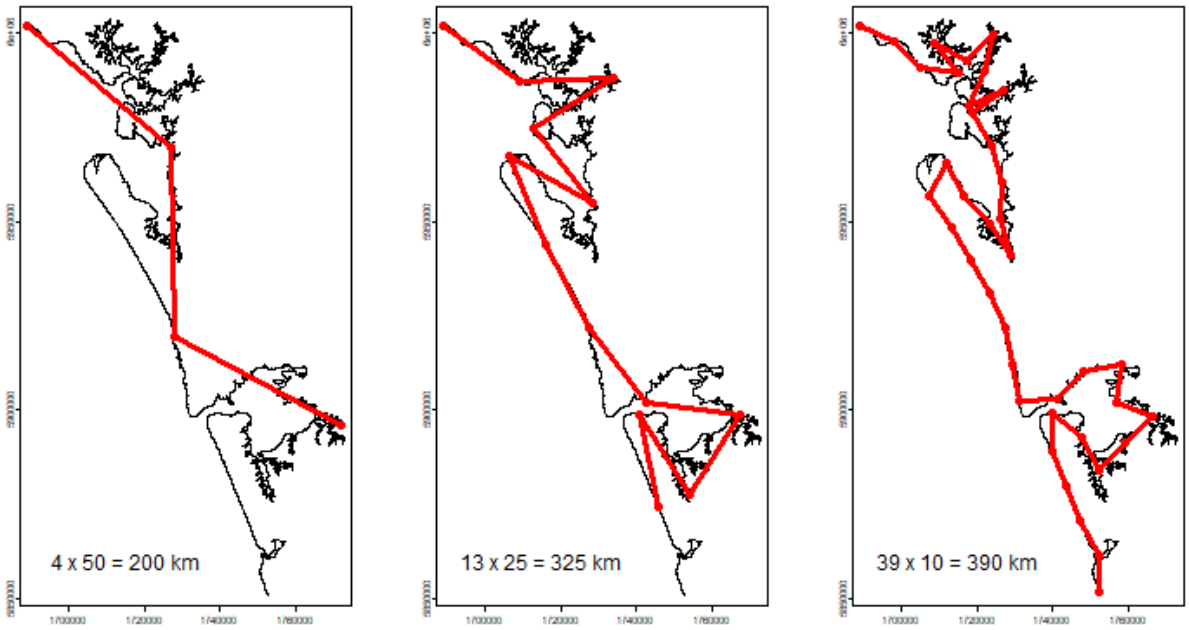
```
stickpoints <- function(x, sticklength, lonlat) {
    nr <- nrow(x)
    pts <- 1
    pt <- 0
    sticklength <- sticklength * 1000
    while(TRUE) {
        pd <- distance(x[1,], x)
        # i is the first point further than the yardstick
        i <- which(pd > sticklength)[1]

        # if we cannot find a point within yardsitck distance we
        # break out of the loop
        if (is.na(i)) break

        # remove the all points we have passed
        x <- x[(i+1):nrow(x), ]
        pt <- pt + i
        pts <- c(pts, pt)
    }
    pts
}
```

With this function we can compute the length of the coastline with yardsticks of different lengths.

```
# get the x and y coordinates of the nodes
g <- as.points(mcoast)
# reverse the order (to start at the top rather than at the bottom)
g <- rev(g)

# three yardstick lengths
sticks <- c(50, 25, 10) # km
# create an empty list for the results
y <- list()
# loop over the yardstick lengths
for (i in 1:length(sticks)) {
    y[[i]] <- stickpoints(g, sticks[i], FALSE)
}
# These last four lines are equivalent to:
# y <- lapply(sticks, function(s) stickpoints(g, s, FALSE))
```

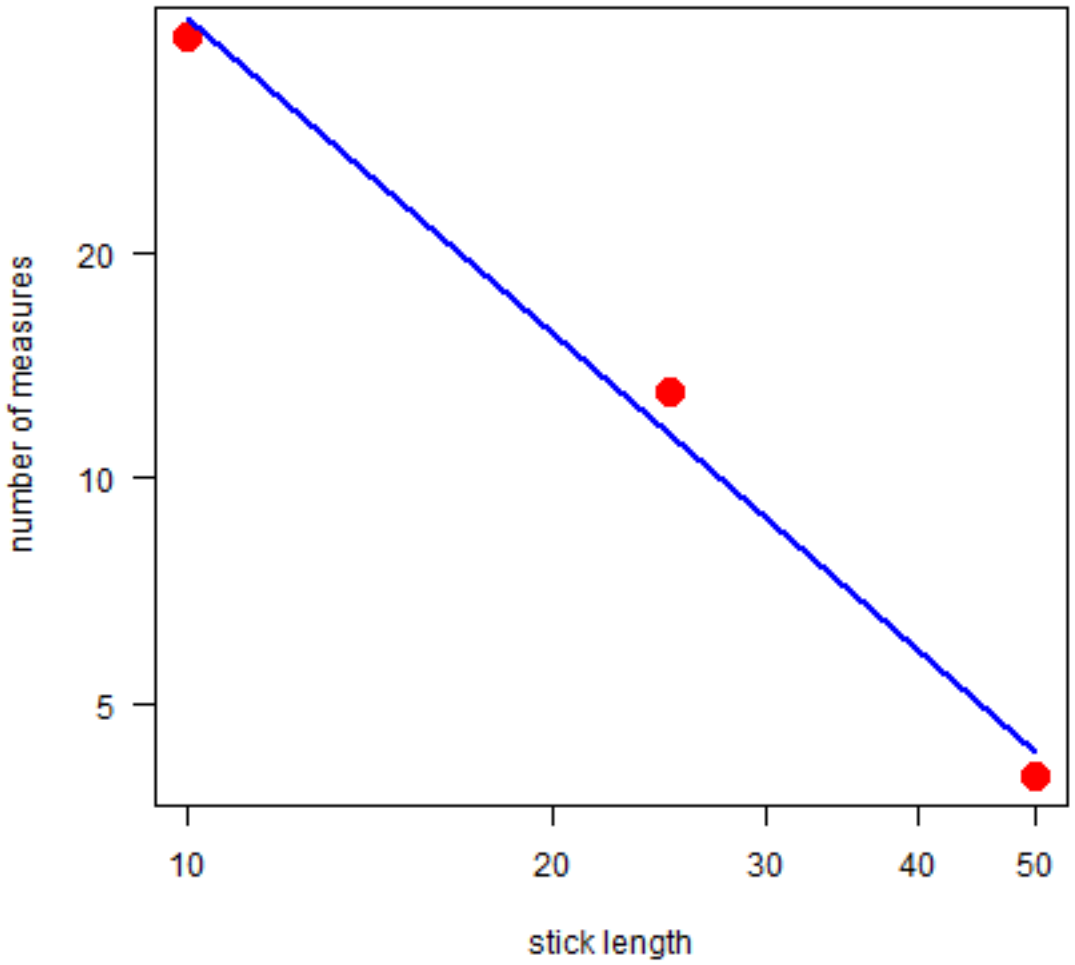
Object y has the indices of g where the stick reached the coastline. We can now make plots as in Figure 1.1. First the first three panels.

```
n <- sapply(y, length)
par(mfrow=c(1,3))
for (i in 1:length(y)) {
    plot(mcoast)
    stops <- y[[i]]
    points(g[stops, ], col="red", pch=20, cex=2)
    lines(g[stops, ], col="red", lwd=3)
    text(1715000, 5860000, paste(n[i], "x", sticks[i], "=", n[i] * sticks[i], "km"),␣
↪cex=1.5)
}
```



The fractal (log-log) plot.

```
plot(sticks, n, log="xy", cex=3, pch=20, col="red",
    xlab="stick length", ylab="number of measures", las=1)
m <- lm(log(n) ~ log(sticks))
lines(sticks, exp(predict(m)), lwd=2, col="blue")
cf <- round(coefficients(m) , 3)
txt <- paste("log N =", cf[2], "log L +", cf[1])
text(6, 222, txt)
```



The fractal dimension D is the (absolute value of the) slope of the regression line.

```
-cf[2]
## log(sticks)
##       1.404
```

Pretty close to the 1.44 that OSU found.

Question 1: *Compare the results in OSU and computed here for the three yardsticks. How and why are they different?*

For a more detailed and complex example, see the fractal dimension of the coastline of Britain page.

CHAPTER

THREE

PITFALLS AND POTENTIAL

3.1 Introduction

This page shows how you can implement the examples provided in Chapter 2 of O'Sullivan and Unwin (2010). To get most out of this, go through the examples slowly, line by line. You should inspect the objects created and read the help files associated with the functions used.

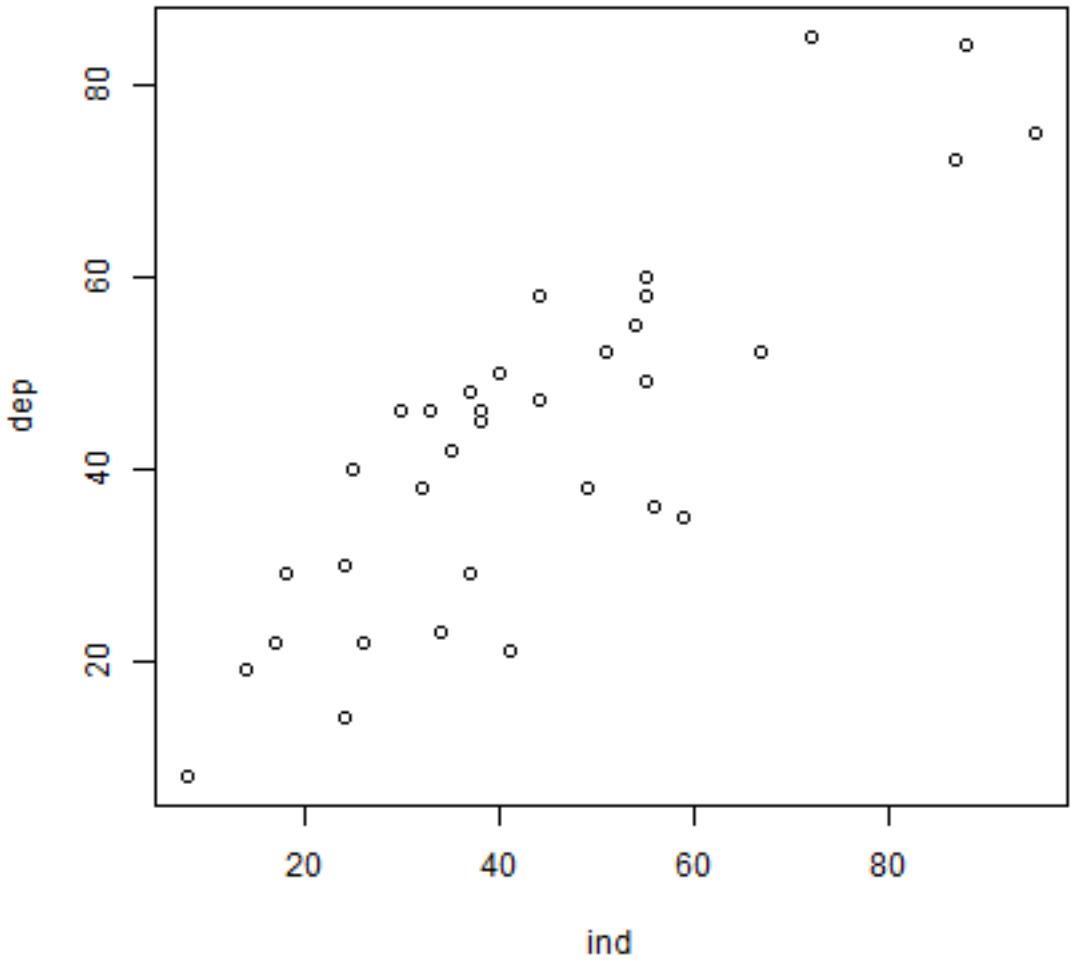
3.2 The Modifiable Areal Unit Problem

Below we recreate the data shown on page 37. There is one region that is divided into 6 x 6 = 36 grid cells. For each cell we have values for two variables. These gridded data can be represented as a matrix, but the easiest way to enter the values is to use a vector (which we can transform to a matrix later). I used line breaks for ease of comparison with the book such that it looks like a matrix anyway.

```
# independent variable
ind <- c(87, 95, 72, 37, 44, 24,
         40, 55, 55, 38, 88, 34,
         41, 30, 26, 35, 38, 24,
         14, 56, 37, 34,  8, 18,
         49, 44, 51, 67, 17, 37,
         55, 25, 33, 32, 59, 54)

# dependent variable
dep <- c(72, 75, 85, 29, 58, 30,
         50, 60, 49, 46, 84, 23,
         21, 46, 22, 42, 45, 14,
         19, 36, 48, 23,  8, 29,
         38, 47, 52, 52, 22, 48,
         58, 40, 46, 38, 35, 55)
```

Now that we have the values, we can make a scatter plot.

```
plot(ind, dep)
```



And here is how you can fit a linear regression model using the `glm` function. `dep ~ ind` means *'dep' is a function of 'ind'*.

```
m <- glm(dep ~ ind)
```

Now let's look at our model `m`.

```
m
##
## Call:  glm(formula = dep ~ ind)
##
## Coefficients:
## (Intercept)          ind
##     10.3750       0.7543
##
## Degrees of Freedom: 35 Total (i.e. Null);  34 Residual
## Null Deviance:       12080
```

(continues on next page)

(continued from previous page)

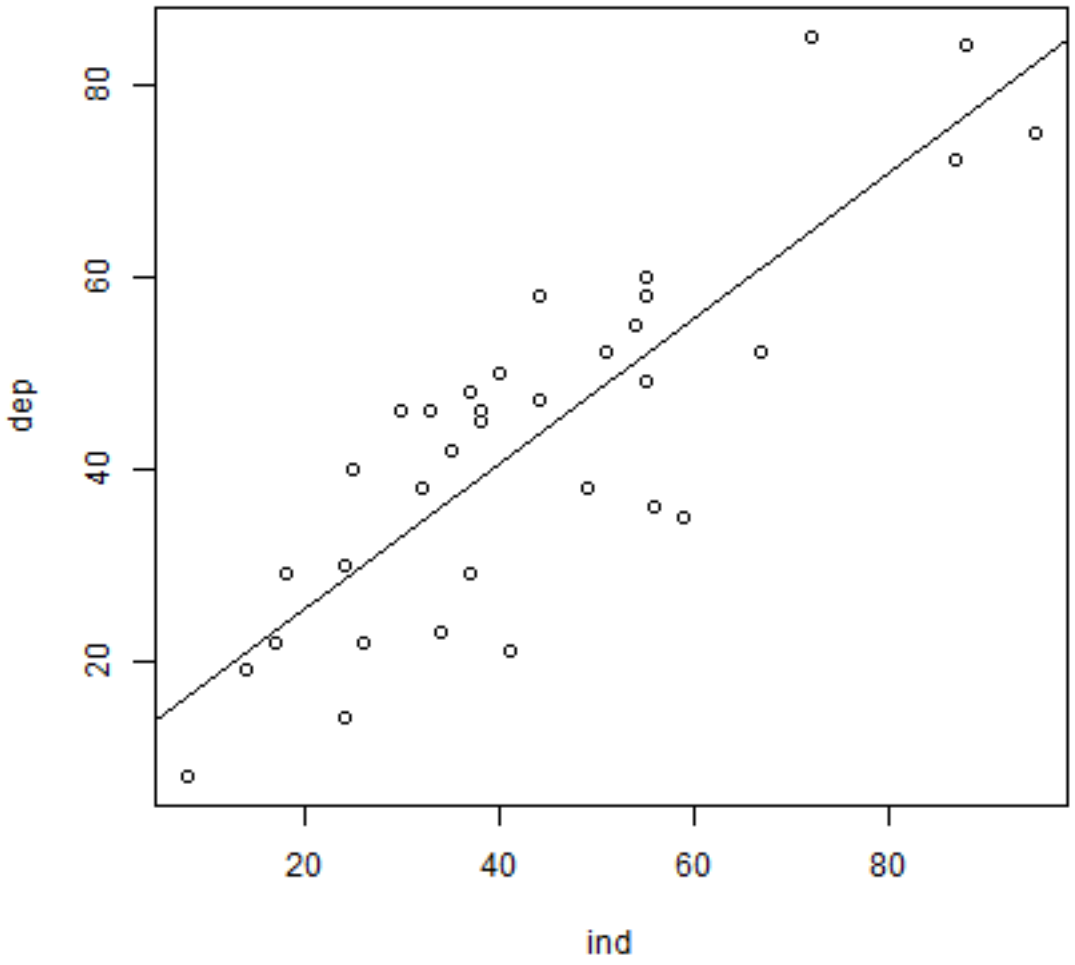
```
## Residual Deviance: 3742  AIC: 275.3
```

To get a bit more information about `m`, we can use the `summary` function.

```
s <- summary(m)
s
##
## Call:
## glm(formula = dep ~ ind)
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.37497    4.12773   2.513   0.0169 *
## ind          0.75435    0.08668   8.703 3.61e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 110.0631)
##
##     Null deviance: 12078.8  on 35  degrees of freedom
## Residual deviance:  3742.1  on 34  degrees of freedom
## AIC: 275.34
##
## Number of Fisher Scoring iterations: 2
```

We can use `m` to add a regression line to our scatter plot.

```
plot(ind, dep)
abline(m)
```



OK. But let's see how to make a plot that looks more like the one in the book. I first set up a plot without axes, and then add the two axes I want (in stead of the standard box). `las=1` rotates the labels to a horizontal position. The arguments `yaxs="i"`, and `xaxs="i"` force the axes to be drawn at the edges of the plot window (overwriting the default to enlarge the ranges by 6%). To get the filled diamond symbol, I use `pch=18`. See `plot(1:25, pch=1:25)` for more numbered symbols.

Then I add the formula by extracting the coefficients from the regression summary object `s` that was created above, and by concatenating the text elements with the `paste0` function. Creating the superscript in R^2 also takes some fiddling. Don't worry about understanding the details of that. There are a few alternative ways to do this, all of them can be found on-line, so there is no need to remember how to do it.

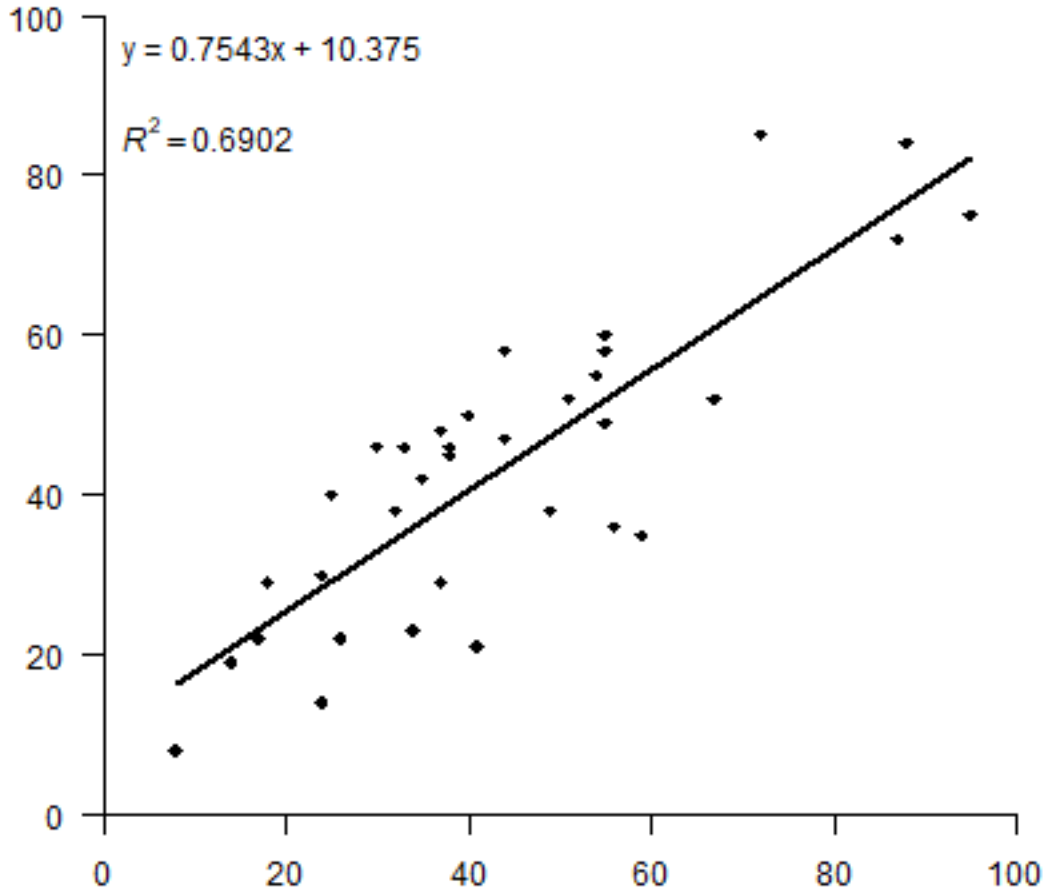
The regression line should only cover the range (min to max value) of variable `ind`. An easy way to do that is to use the regression model to predict values for these extremes and draw a line between these.

```
plot(ind, dep, pch=18, xlim=c(0,100), ylim=c(0,100),
    axes=FALSE, xlab='', ylab='', yaxs="i", xaxs="i")
axis(1, at=(0:5)*20)
axis(2, at=(0:5)*20, las=1)
```

(continues on next page)

(continued from previous page)

```
# create regression formula
f <- paste0('y = ', round(s$coefficients[2], 4), 'x + ', round(s$coefficients[1], 4))
# add the text in variable f to the plot
text(0, 96, f, pos=4)
# compute r-squared
R2 <- cor(dep, predict(m))^2

# set up the expression (a bit complex, this)
r2 <- bquote(italic(R)^2 == .(round(R2, 4)))
# and add it to the plot
text(0, 85, r2, pos=4)

# compute regression line
# create a data.frame with the range (minimum and maximum) of values of ind
px <- data.frame(ind = range(ind))
# use the regression model to get predicted value for dep at these two extremes
py <- predict(m, px)
# combine the min and max values and the predicted values
ln <- cbind(px, py)
# add to the plot as a line
lines(ln, lwd=2)
```



Now the aggregation. I first turn the vectors into matrices, which is very easy to do. You should play with the matrix function a bit to see how it works. It is particularly important that you understand the argument `byrow=TRUE`. By default `R` fills matrices column-wise.

```
mi <- matrix(ind, ncol=6, nrow=6, byrow=TRUE)
md <- matrix(dep, ncol=6, nrow=6, byrow=TRUE)
```

Question 1: *Create these matrices from* `ind` *and* `dep` *without using* `byrow=TRUE`. *Hint: use the* `t` *function after you made the matrix.*

The type of aggregation as shown in Figure 2.1 is not a very typical operation in the context of matrix manipulation. However, it is very common to do this with raster data. So let's first transform the matrices to objects that represent raster data, `SpatRaster` objects in this case. This class is defined in the `terra` package, so we need to load that first. If `library(terra)` gives this error: `Error in library("terra") : there is no package called 'terra'` you need to install the package first, using this command: `install.packages("terra")`.

```
# load package
```

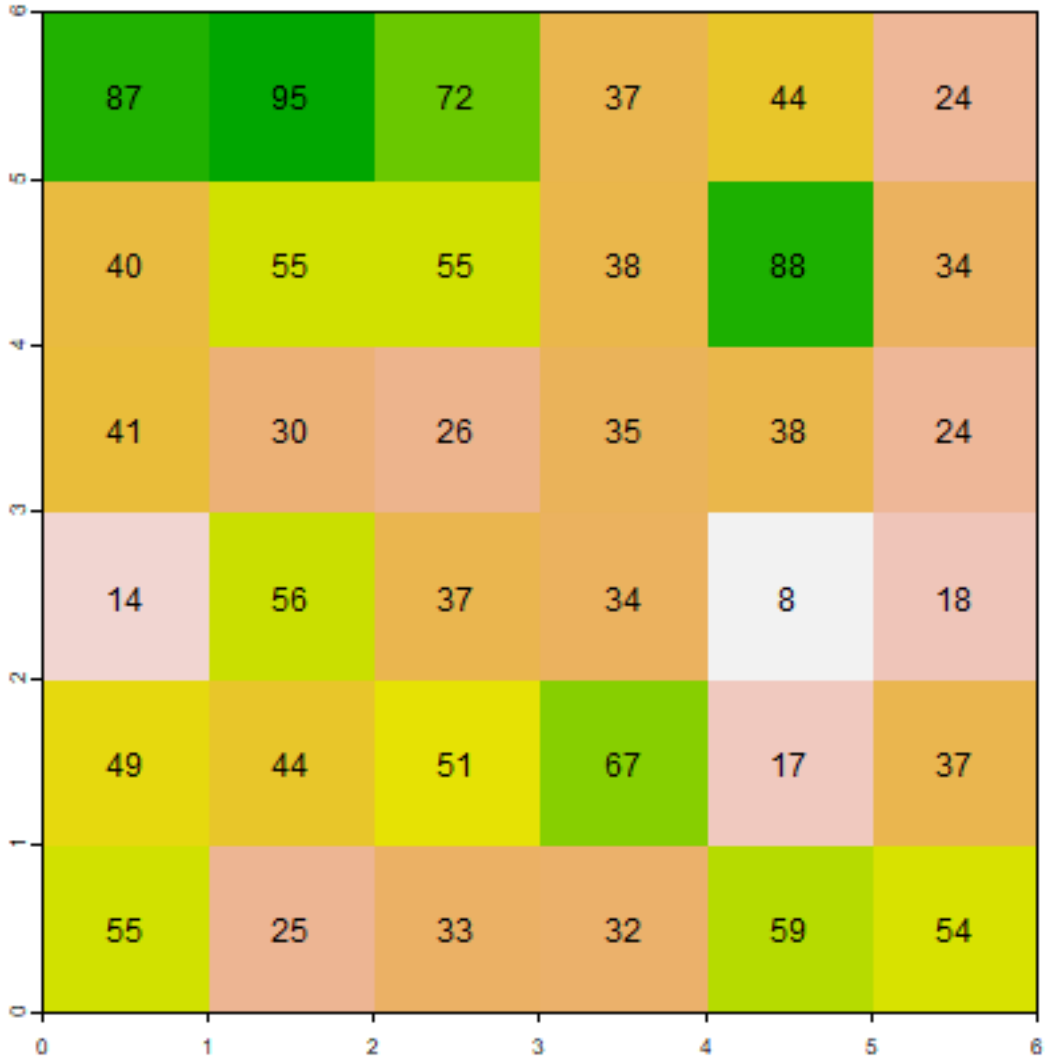
(continues on next page)

(continued from previous page)

```
library(terra)
## terra 1.7.62

# turn matrices into SpatRaster objects
ri <- rast(mi)
rd <- rast(md)
```

Inspect one of these new objects

```
ri
## class       : SpatRaster
## dimensions  : 6, 6, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0, 6, 0, 6  (xmin, xmax, ymin, ymax)
## coord. ref. :
## source(s)   : memory
## name        : lyr.1
## min value   :     8
## max value   :    95
plot(ri, legend=FALSE)
text(ri)
```



The `raster` package has an aggregate function that we will use. We specify that we want to aggregate sets of 2 columns, but not aggregate rows. The values for the new cells should be computed from the original cells using the mean function.

Question 2: *Instead of the ``mean`` function What other functions could, in principle, reasonably be used in an aggregation of raster cells?*

```
ai1 <- aggregate(ri, c(2, 1), fun=mean)
ad1 <- aggregate(rd, c(2, 1), fun=mean)
```

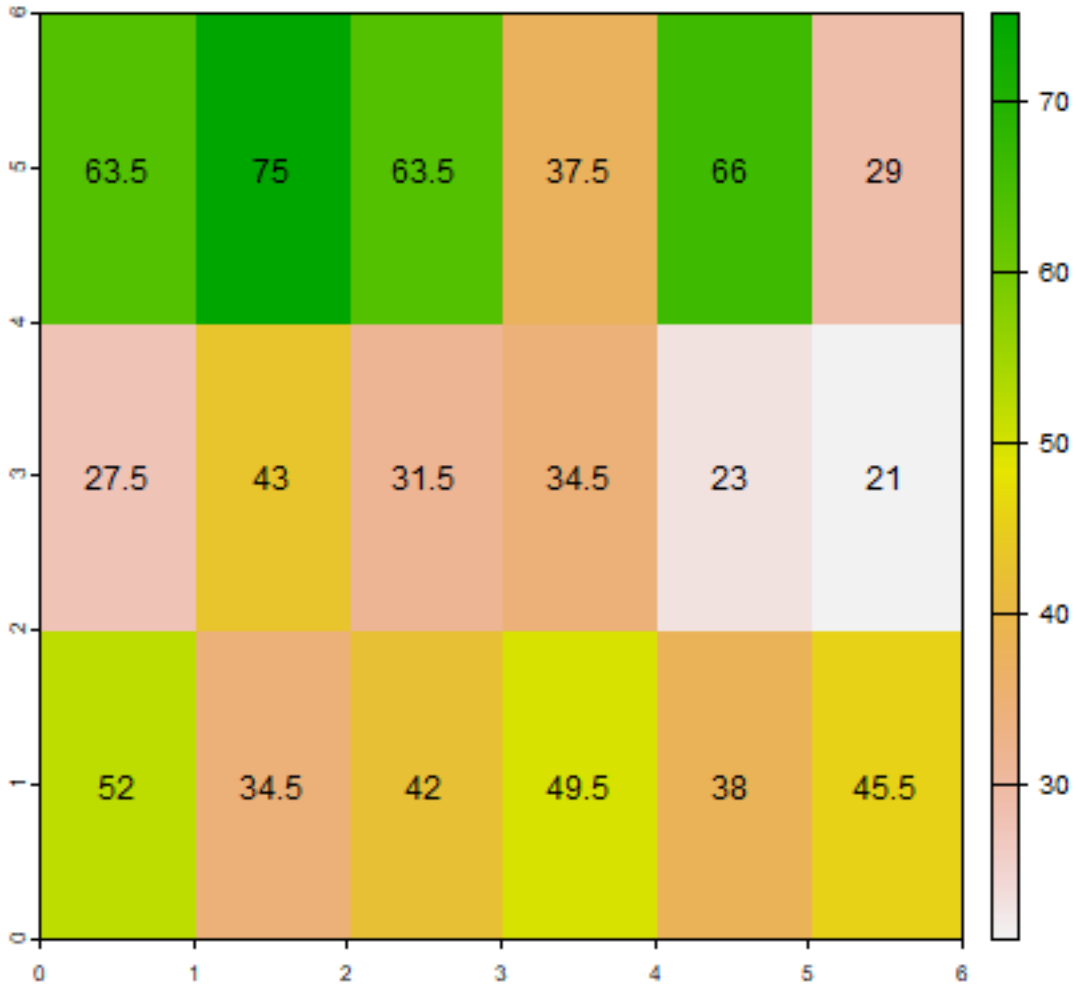
Inspect the results

```
as.matrix(ai1)
##       lyr.1
##  [1,]  63.5
##  [2,]  75.0
##  [3,]  63.5
```

(continues on next page)

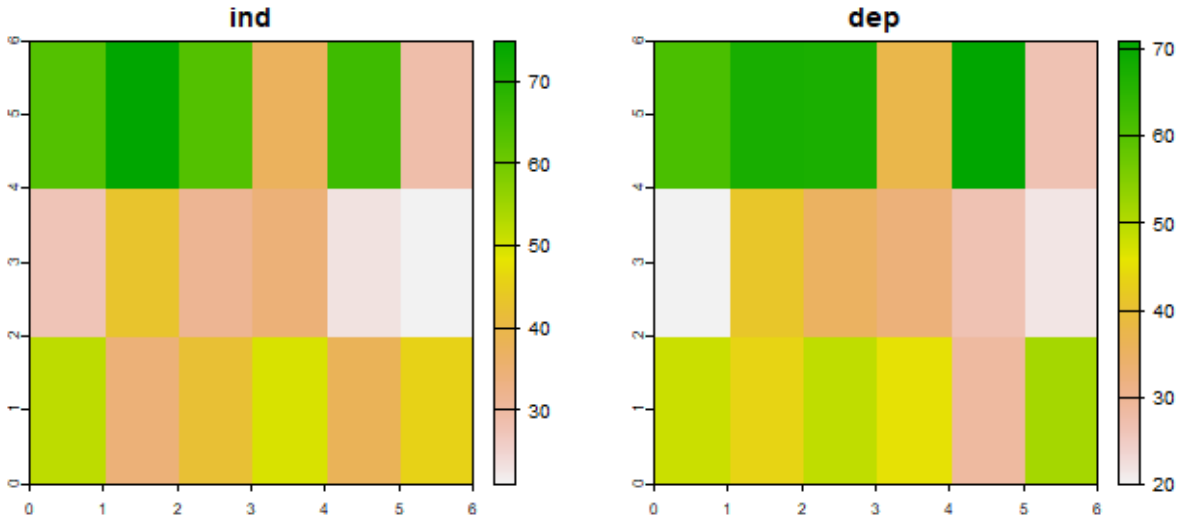
(continued from previous page)

```
##  [4,]  37.5
##  [5,]  66.0
##  [6,]  29.0
##  [7,]  27.5
##  [8,]  43.0
##  [9,]  31.5
## [10,]  34.5
## [11,]  23.0
## [12,]  21.0
## [13,]  52.0
## [14,]  34.5
## [15,]  42.0
## [16,]  49.5
## [17,]  38.0
## [18,]  45.5
plot(ai1)
text(ai1, digits=1)
```



To be able to do the regression as we did above, I first combine the two `SpatRaster` objects into a (multi-layer) object.

```
s1 <- c(ai1, ad1)
names(s1) <- c("ind", "dep")
s1
## class       : SpatRaster
## dimensions  : 3, 6, 2  (nrow, ncol, nlyr)
## resolution  : 1, 2  (x, y)
## extent      : 0, 6, 0, 6  (xmin, xmax, ymin, ymax)
## coord. ref. :
## source(s)   : memory
## names       : ind, dep
## min values  :  21,  20
## max values  :  75,  71
plot(s1)
```



Below I coerce the `SpatRaster` into a `data.frame`. In R, most functions for statistical analysis want the input data as a `data.frame`.

```
d1 <- as.data.frame(s1)
head(d1)
##    ind  dep
## 1 63.5 61.0
## 2 75.0 67.5
## 3 63.5 67.0
## 4 37.5 37.5
## 5 66.0 71.0
## 6 29.0 26.5
```

To recap: each matrix was used to create a `SpatRaster` that we aggregated and then combined. Each of the aggregated `SpatRaster` layers became a single variable (column) in the `data.frame`. If would perhaps have been more efficient to first make a `SpatRaster` and then `aggregate`.

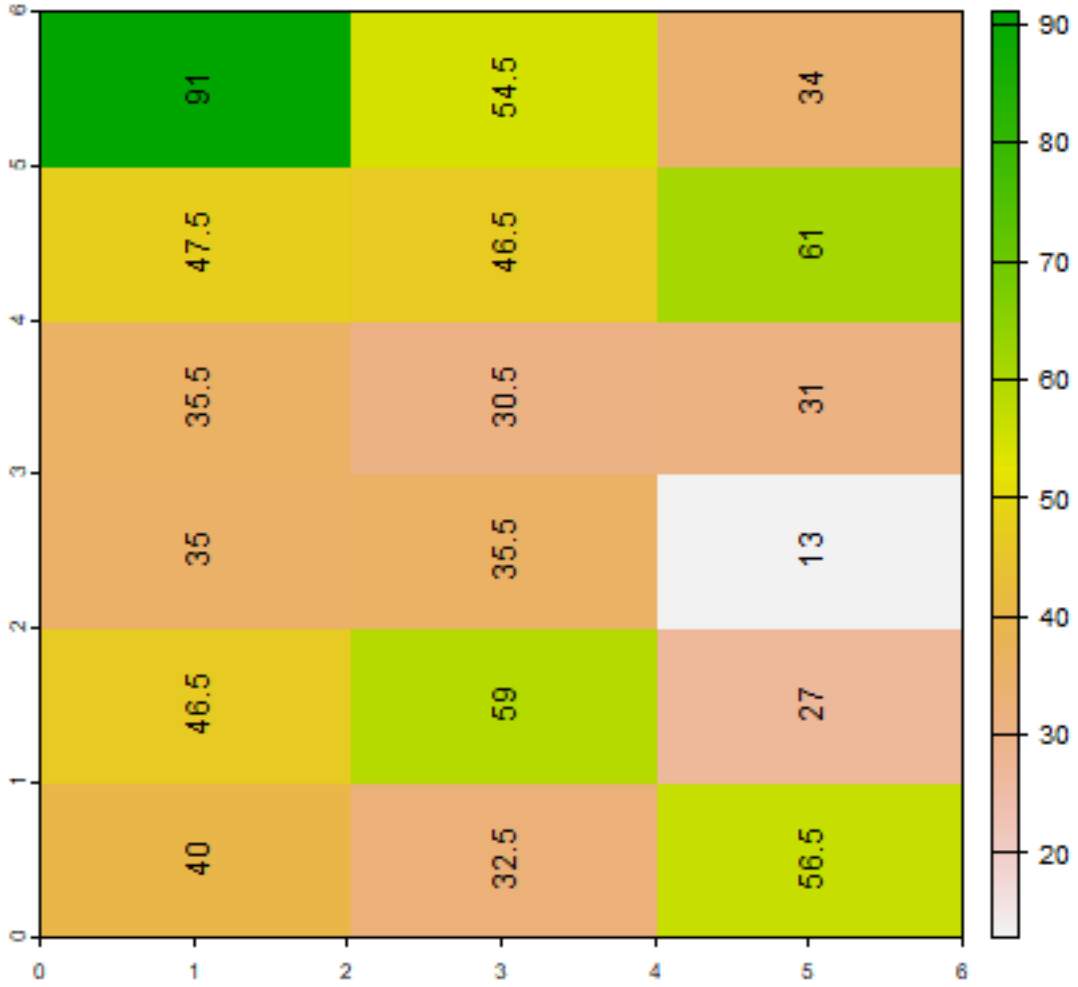
Question 3: *There are other ways to do the above (converting two* `SpatRaster` *objects to a* `data.frame`*). Show how to obtain the same result (*`d1`*) using* `as.vector` *and* `cbind`*.*

Let's fit a regression model again, now with these aggregated data:

```
ma1 <- glm(dep~ind, data=d1)
```

Same idea for for the other aggregation ('Aggregation scheme 2'). But note that the arguments to the `aggregate` function are, of course, different.

```
ai2 <- aggregate(ri, c(1, 2), fun=mean)
ad2 <- aggregate(rd, c(1, 2), fun=mean)
plot(ai2)
text(ai2, digits=1, srt=90)
```




```
s2 <- c(ai2, ad2)
names(s2) <- c('ind', 'dep')
# coerce to data.frame
d2 <- as.data.frame(s2)
ma2 <- glm(dep ~ ind, data=d2)
```

Now we have three regression model objects. We first created object `m`, and then the two models with aggregated data: `ma1` and `ma2`. Compare the regression model coefficients.

```
m$coefficients
## (Intercept)         ind
##  10.3749675   0.7543472
ma1$coefficients
## (Intercept)         ind
##   1.2570386   0.9657093
ma2$coefficients
```

(continues on next page)

(continued from previous page)

```
## (Intercept)         ind
##  13.5899183   0.6798216
```

Re-creating figure 2.1 takes some effort. We want to make a similar figure three times (two matrices and a plot). That makes it efficient and practical to use a function. Look here if you do not remember how to write and use your own function in R:

The function I wrote, called `plotMAUP`, is a bit complex, so I do not show it here. But you can find it in the source code for this page. Have a look at it if you can, don't worry about the details, but see if you can understand the main reason for each step. It helps to try the lines of the function one by one (outside of the function).

To use the `plotMAUP` function, I first set up a plotting canvas of 3 rows and 3 columns, using the `mfrow` argument in the `par` function. The `par` function is very important for customizing plots — and it has an overwhelming number of options to consider. See `?par`. The `mai` argument is used to change the margins around each plot.

```
# plotting parameters
par(mfrow=c(3,3), mai=c(0.25,0.15,0.25,0.15))

# Now call plotMAUP 3 times
plotMAUP(ri, rd, title=c('Independent variable', 'Dependent variable'))
# aggregation scheme 1
plotMAUP(ai1, ad1, title='Aggregation scheme 1')
# aggregation scheme 2
plotMAUP(ai2, ad2, title='Aggregation scheme 2')
```

3.3 Distance, adjacency, interaction, neighborhood

Here we explore the data in Figure 2.2 (page 46). The values used are not exactly the same (as they were not provided in the text), but it is all very similar.

Set up the data, using x-y coordinates for each point:

```
A <- c(40, 43)
B <- c(1, 101)
C <- c(54, 111)
D <- c(104, 65)
E <- c(60, 22)
F <- c(20, 2)
pts <- rbind(A,B,C,D,E,F)
head(pts)
##   [,1] [,2]
## A   40   43
## B    1  101
## C   54  111
## D  104   65
## E   60   22
## F   20    2
```

Plot the points and labels:

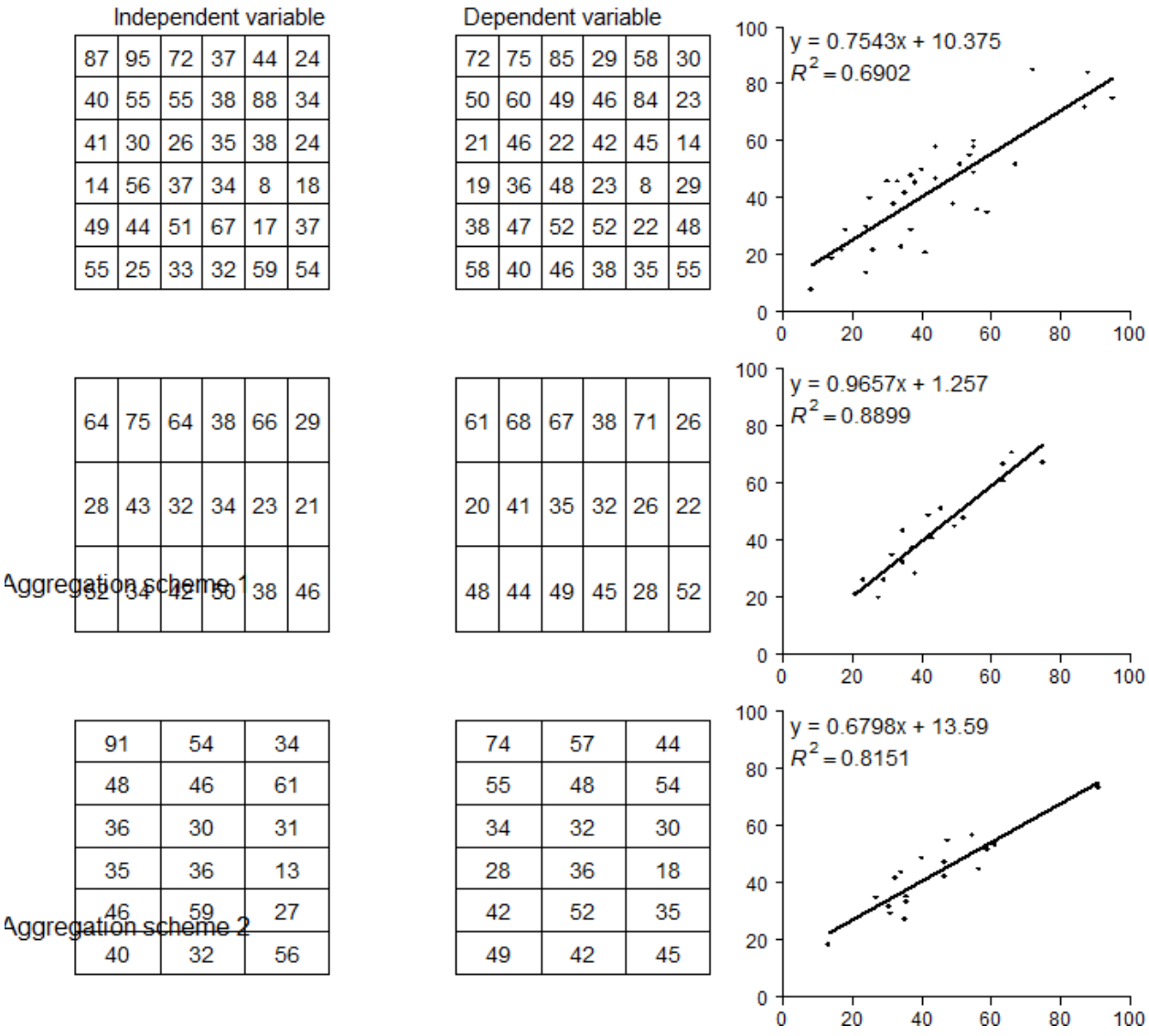
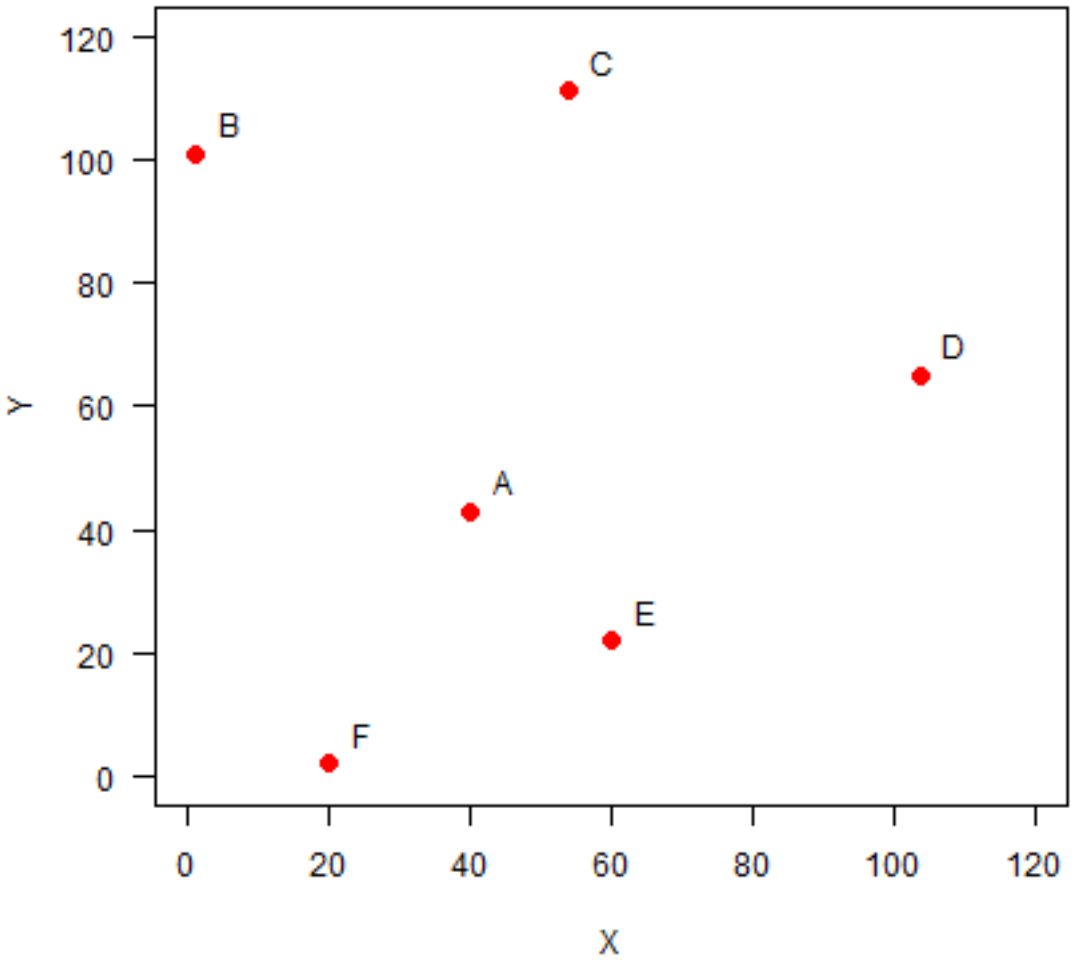


Fig. 1: Figure 2.1 An illustration of MAUP

```
plot(pts, xlim=c(0,120), ylim=c(0,120), pch=20, cex=2, col='red', xlab='X', ylab='Y',␣
↪las=1)
text(pts+5, LETTERS[1:6])
```



3.3.1 Distance

It is easy to make a distance matrix (see page 47)

```
dis <- dist(pts)
dis
##           A         B         C         D         E
## B  69.89278
## C  69.42622  53.93515
## D  67.67570 109.11004  67.94115
## E  29.00000  98.60020  89.20202  61.52235
```

(continues on next page)

(continued from previous page)

```
## F  45.61798 100.80675 114.17968 105.00000  44.72136
D <- as.matrix(dis)
round(D)
##     A   B   C   D  E   F
## A   0  70  69  68 29  46
## B  70   0  54 109 99 101
## C  69  54   0  68 89 114
## D  68 109  68   0 62 105
## E  29  99  89  62  0  45
## F  46 101 114 105 45   0
```

Distance matrices are used in all kinds of non-geographical applications. For example, they are often used to create cluster diagrams (dendograms).

Question 4: *Show R code to make a cluster dendogram summarizing the distances between these six sites, and plot it. See* `?hclust`.

3.3.2 Adjacency

Distance based adjacency

To get the adjacency matrix, here defined as points within a distance of 50 from each other is trivial given that we have the distances `D`.

```
a <-  D < 50
a
##       A     B     C     D     E     F
## A  TRUE FALSE FALSE FALSE  TRUE  TRUE
## B FALSE  TRUE FALSE FALSE FALSE FALSE
## C FALSE FALSE  TRUE FALSE FALSE FALSE
## D FALSE FALSE FALSE  TRUE FALSE FALSE
## E  TRUE FALSE FALSE FALSE  TRUE  TRUE
## F  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

To make this match matrix 2.6 on page 48, set the diagonal values to `NA` (we do not consider a point to be adjacent to itself). Also change the change the `TRUE/FALSE` values to to `1/0` using a simple trick (multiplication with 1)

```
diag(a) <- NA
adj50 <- a * 1
adj50
##    A  B  C  D  E  F
## A NA  0  0  0  1  1
## B  0 NA  0  0  0  0
## C  0  0 NA  0  0  0
## D  0  0  0 NA  0  0
## E  1  0  0  0 NA  1
## F  1  0  0  0  1 NA
```

Three nearest neighbors

Computing the "three nearest neighbors" adjacency-matrix requires a bit more advanced understanding of R.

For each row, we first get the column numbers in order of the values in that row (that is, the numbers indicate how the values are ordered).

```
cols <- apply(D, 1, order)
# we need to transpose the result
cols <- t(cols)
```

And then get columns 2 to 4 (why not column 1?)

```
cols <- cols[, 2:4]
cols
##   [,1] [,2] [,3]
## A    5    6    4
## B    3    1    5
## C    2    4    1
## D    5    1    3
## E    1    6    4
## F    5    1    2
```

As we now have the column numbers, we can make the row-column pairs that we want (`rowcols`).

```
rowcols <- cbind(rep(1:6, each=3), as.vector(t(cols)))
head(rowcols)
##      [,1] [,2]
## [1,]    1    5
## [2,]    1    6
## [3,]    1    4
## [4,]    2    3
## [5,]    2    1
## [6,]    2    5
```

We use these pairs as indices to change the values in matrix `Ak3`.

```
Ak3 <- adj50 * 0
Ak3[rowcols] <- 1
Ak3
##    A  B  C  D  E  F
## A NA  0  0  1  1  1
## B  1 NA  1  0  1  0
## C  1  1 NA  1  0  0
## D  1  0  1 NA  1  0
## E  1  0  0  1 NA  1
## F  1  1  0  0  1 NA
```

Weights matrix

Getting the weights matrix is simple.

```
W <- 1 / D
round(W, 4)
##        A      B      C      D      E      F
## A    Inf 0.0143 0.0144 0.0148 0.0345 0.0219
## B 0.0143    Inf 0.0185 0.0092 0.0101 0.0099
## C 0.0144 0.0185    Inf 0.0147 0.0112 0.0088
## D 0.0148 0.0092 0.0147    Inf 0.0163 0.0095
## E 0.0345 0.0101 0.0112 0.0163    Inf 0.0224
## F 0.0219 0.0099 0.0088 0.0095 0.0224    Inf
```

Row-normalization is not that difficult either. First get rid if the `Inf` values by changing them to `NA`. (Where did the `Inf` values come from?)

```
W[!is.finite(W)] <- NA
```

Then compute the row sums.

```
rtot <- rowSums(W, na.rm=TRUE)
# this is equivalent to
# rtot <- apply(W, 1, sum, na.rm=TRUE)
rtot
##          A          B          C          D          E          F
## 0.09989170 0.06207541 0.06763182 0.06443810 0.09445017 0.07248377
```

Divide the rows by their totals and check if they row sums add up to 1.

```
W <- W / rtot
rowSums(W, na.rm=TRUE)
## A B C D E F
## 1 1 1 1 1 1
```

The values in the columns do not add up to 1.

```
colSums(W, na.rm=TRUE)
##         A         B         C         D         E         F
## 1.3402904 0.8038417 0.9108116 0.8166821 1.2350790 0.8932953
```

Question 5: *Show how you can do 'column-normalization' (Just an exercise, in spatial data analysis this is not a typical thing to do).*

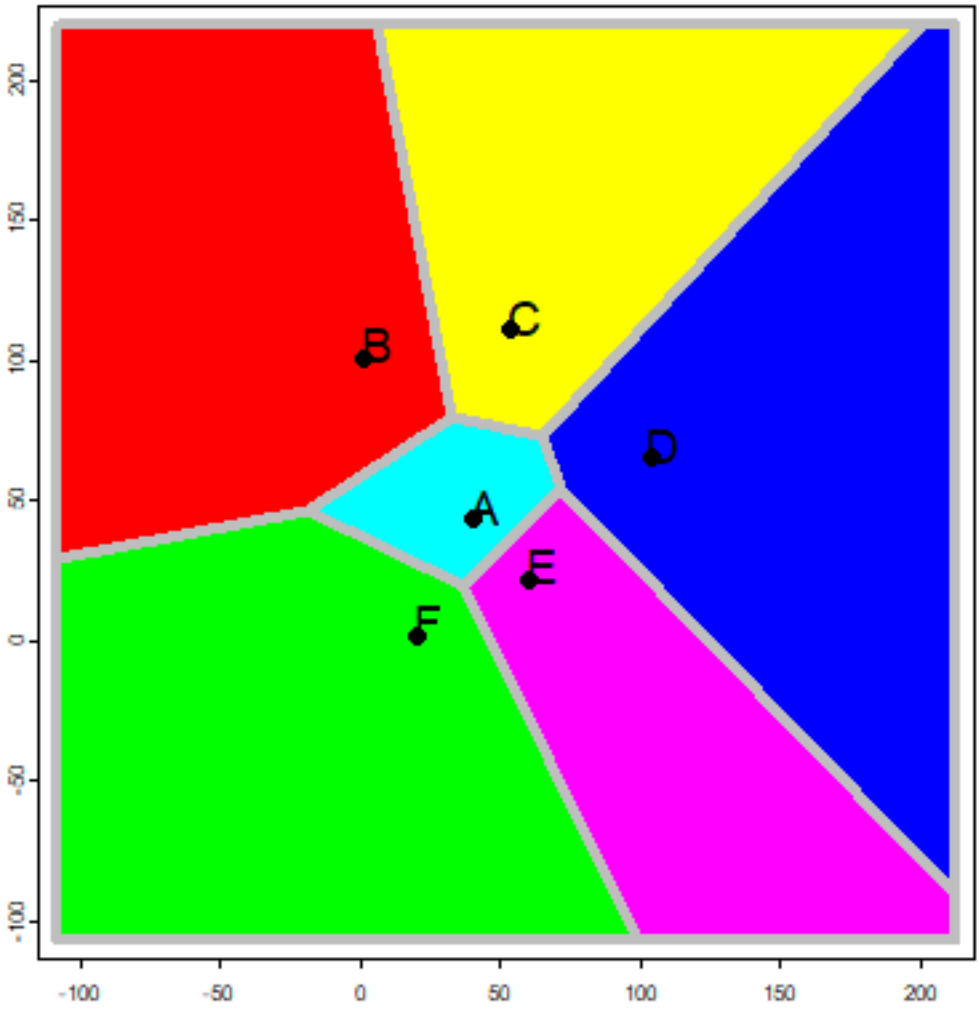
3.3.3 Proximity polygons

Proximity polygons are discussed on pages 50-52. Here I show how you can compute these with the `voronoi` function. We use the data from the previous example.

```
v <- voronoi(vect(pts))
```

Here is a plot of our proximity polygons (also known as a Voronoi diagram).

```
par(mai=rep(0,4))
plot(v, lwd=4, border='gray', col=rainbow(6))
points(pts, pch=20, cex=2)
text(pts+5, toupper(letters[1:6]), cex=1.5)
```



Note that the `voronoi` functions returns a `SpatVector`. This is a class (type of object) that can be used to represent geospatial polygons in *R*.

```
class(v)
## [1] "SpatVector"
## attr(,"package")
## [1] "terra"
v
##  class       : SpatVector
##  geometry    : polygons
##  dimensions  : 6, 0  (geometries, attributes)
```

(continues on next page)

(continued from previous page)

```
##  extent      : -108, 213, -107, 220  (xmin, xmax, ymin, ymax)
##  coord. ref. :
```

CHAPTER

FOUR

FUNDAMENTALS

4.1 Processes and patterns

This handout accompanies Chapter 4 in O'Sullivan and Unwin (2010) by working out the examples in *R*. Figure 4.2 (on page 96) shows values for a deterministic spatial process z = 2x + 3y. Below are two ways to create such a plot in *R*. The first one uses "base" *R*. That is, it does not use explicitly spatial objects (classes). I use the expand.grid function to create a matrix with two columns with all combinations of `0:7` with `0:7`:

```
x <- 0:7
y <- 0:7
xy <- expand.grid(x, y)
colnames(xy) <- c("x", "y")
head(xy)
##   x y
## 1 0 0
## 2 1 0
## 3 2 0
## 4 3 0
## 5 4 0
## 6 5 0
```

Now we can use these values to compute the values of `z` that correspond to the values of `x` (the first column of object `xy`) and `y` (the second column).

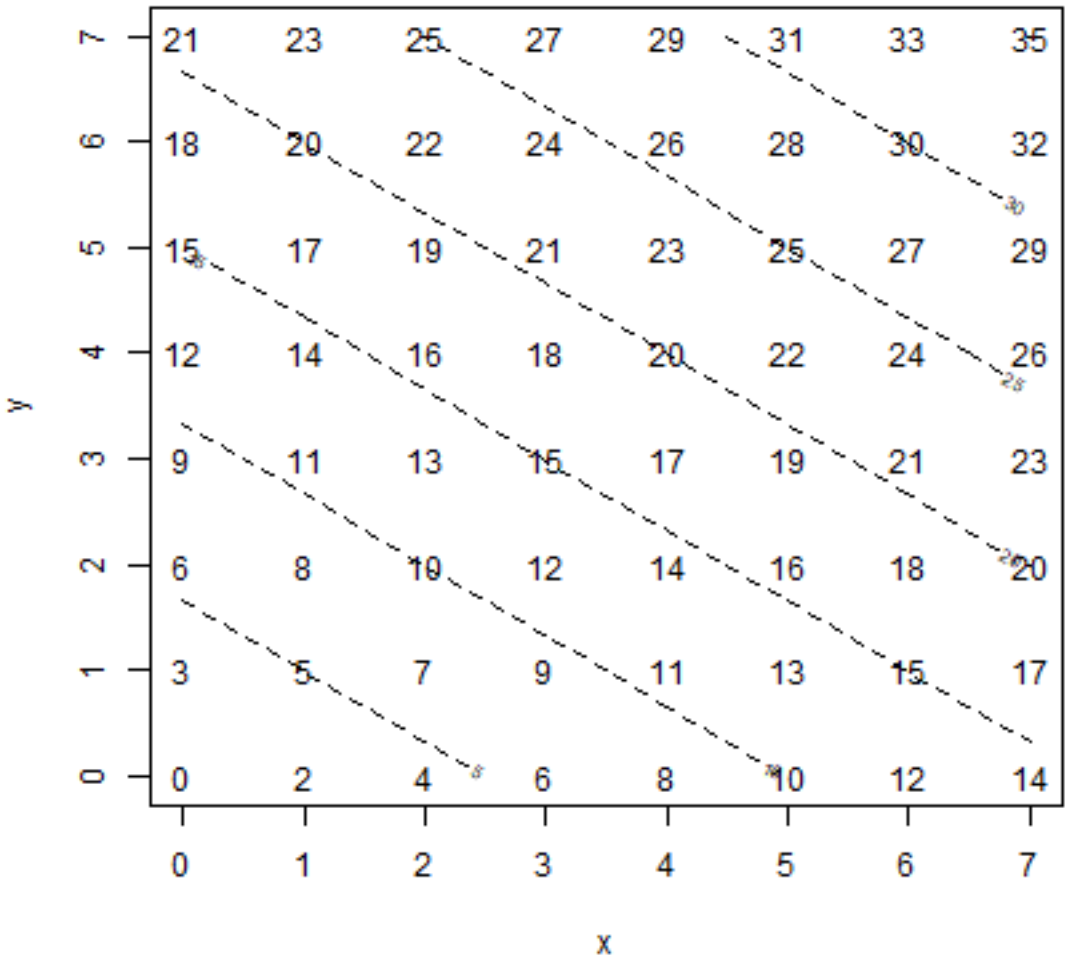
```
z <- 2*xy[,1] + 3*xy[,2]
zm <- matrix(z, ncol=8)
```

Here I do the same thing, but using a *function* of my own making (called `detproc`); just to get used to the idea of writing and using your own functions.

```
detproc <- function(x, y) {
  z <- 2*x + 3*y
  return(z)
}

v <- detproc(xy[,1], xy[,2])
zm <- matrix(v, ncol=8)
```

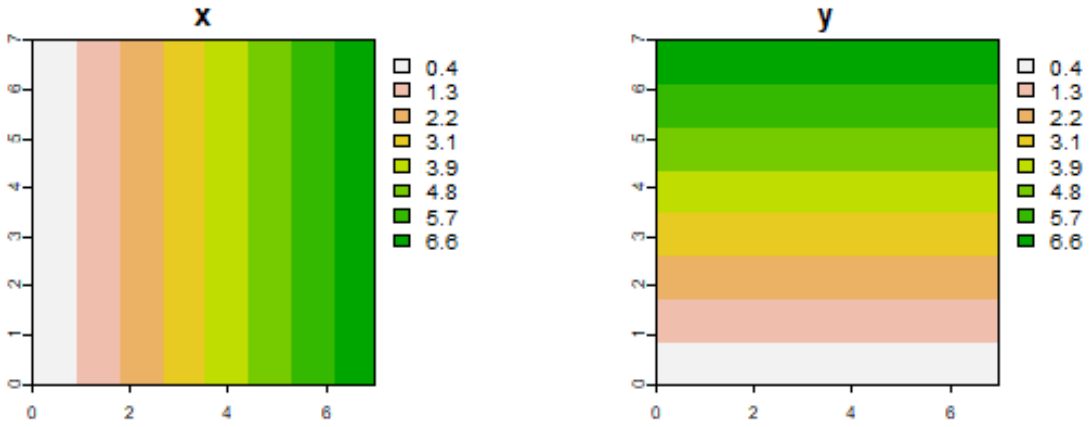
Below, I use a trick `plot(x, y, type="n")` to set up a plot with the correct axes, that is otherwise blank. I do this because I do not want any dots on the plot. Instead of showing dots, I use the 'text' function to add the labels (as in the book).

```
plot(x, y, type="n")
text(xy[,1], xy[,2], z)
contour(x, y, zm, add=TRUE, lty=2)
```



Now, let's do the same thing as above, but now by using a spatial data approach. Instead of a matrix, we use SpatRasters. First we create an empty raster with eight rows and columns, and with x and y coordinates going from 0 to 7. The `init` function sets the values of the cells to either the x or the y coordinate (or something else, see `?init`).

```
library(terra)
## terra 1.7.62
r <- rast(xmin=0, xmax=7, ymin=0, ymax=7, ncol=8, nrow=8)
X <- init(r, "x")
Y <- init(r, "y")
par(mfrow=c(1,2))
plot(X, main="x")
plot(Y, main="y")
```



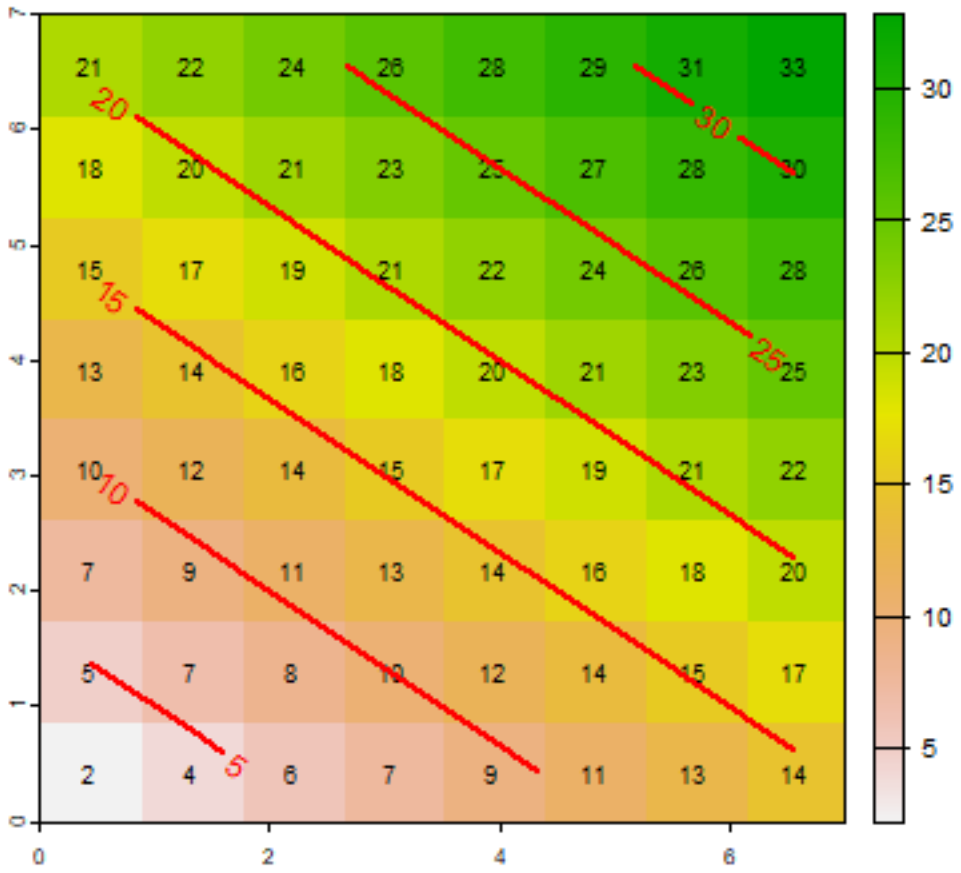
We can use algebraic expressions with SpatRasters

```
Z <- 2*X + 3*Y
```

Do you think it is *possible* to do `Z <- detproc(X, Y)`*?* (try it).

Plot the result.

```
plot(Z)
text(Z, cex=.75)
contour(Z, add=TRUE, labcex=1, lwd=2, col="red")
```



The above does not seem very interesting. But, if the process is complex, a map of the outcome of a deterministic process can actually be very interesting. For example, in ecology and associated sciences there are many "mechanistic" (= process) models that dynamically (= over time) simulate ecosystem processes such as vegetation dynamics and soil greenhouse gas emissions that depend on interaction of the weather, soil type, genotypes, and management; making it hard to predict the model outcome over space. In this context, stochasticity still exists through the input variables such as rainfall. Many other models exist that have a deterministic and stochastic components (for example, global climate models).

Below I follow the book by adding a stochastic element to the deterministic process by adding variable `r` to the equation: `z = 2x + 3y + r`; where r is a random value that can be -1 or +1. Many examples in *R* manuals use randomly generated values to illustrate how a particular function work. But much real data analysis also depends on randomly selected variables, for example, to create a "null model" (such as CSR) to compare with an observed data set.

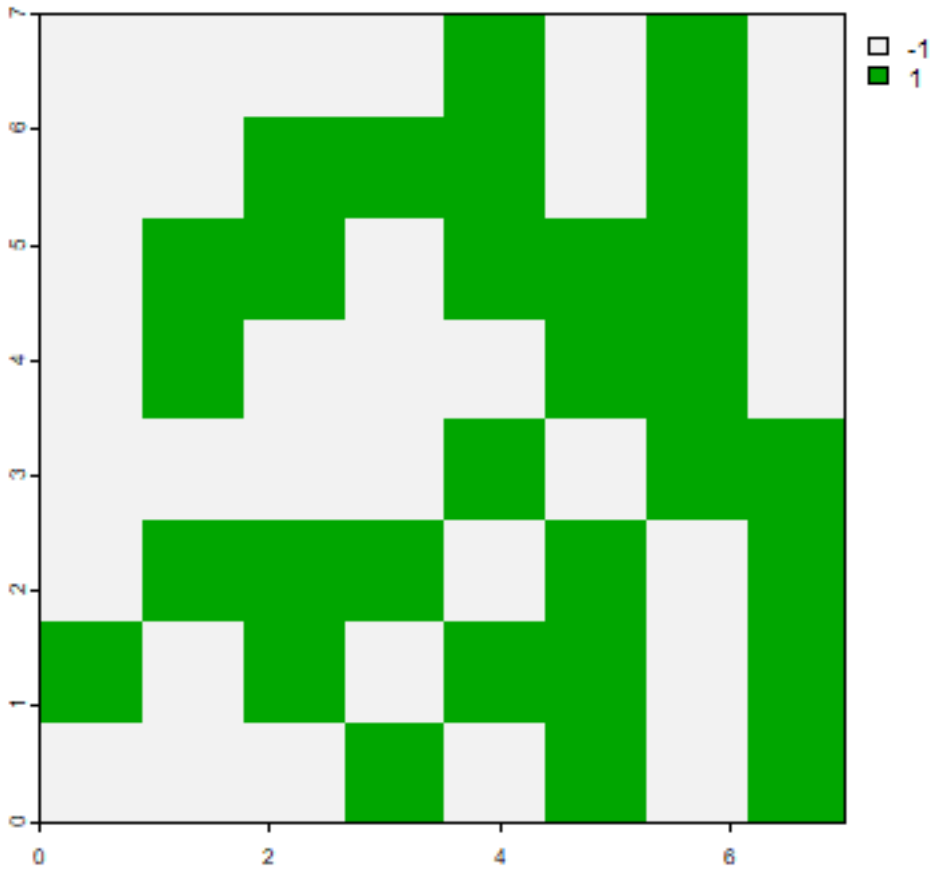
There are different ways to get random numbers, and you should pay attention to that. The `sample` function returns randomly selected values from a set that you provide (by default this is done without replacement). We need a random value for each cell of SpatRaster `r`, and assign these to a new SpatRaster with the same properties (spatial extent and resolution). When you work with random values, the results will be different each time you run some code (that is the point); but sometimes it is desireable to recreate exactly the same random sequence. The function `set.seed` allows you to do that — to create the same random sequence over and over again.

```
set.seed(987)
s <- sample(c(-1, 1), ncell(r), replace=TRUE)
s[1:8]
## [1] -1 -1 -1 -1  1 -1  1 -1
```

(continues on next page)

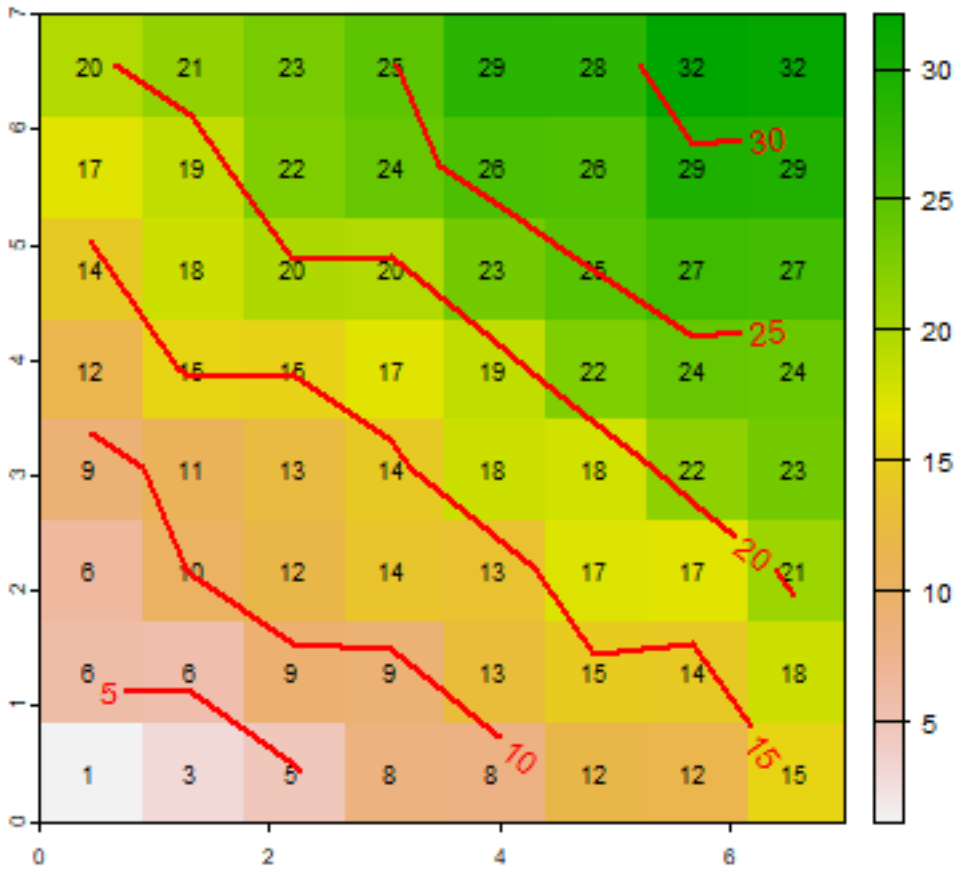
(continued from previous page)

```
R <- setValues(r, s)
plot(R)
```



Now we can solve the formula and look at the result

```
Z <- 2*X + 3*Y + R
plot(Z)
text(Z, cex=.75)
contour(Z, add=T, labcex=1, lwd=2, col="red")
```

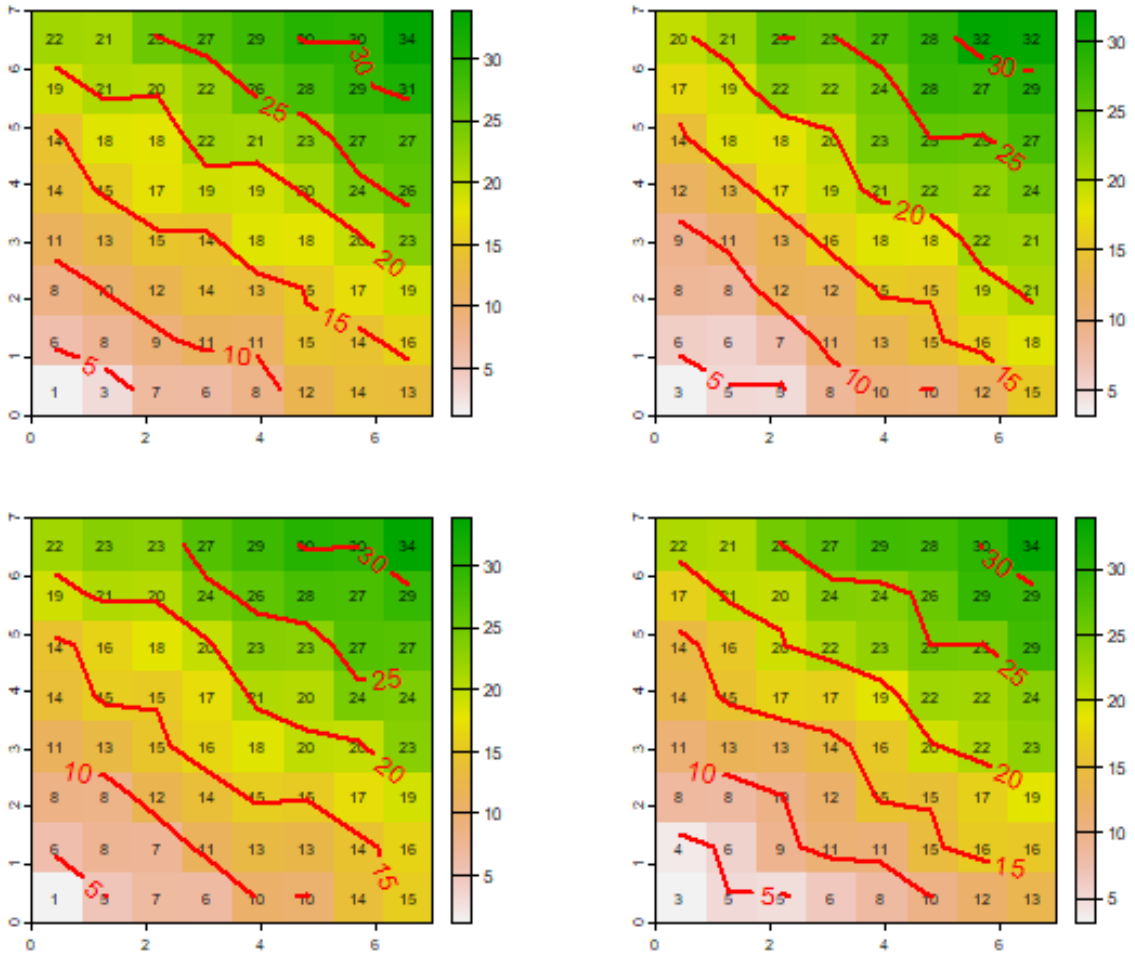


The figure above is a pattern from a (partly) random process. The process can generate other patterns, as is shown below. Because we want to repeat the same thing (process, code) a number of times, it is convenient to define a (pattern generating) function.

```
f <- function() {
    s <- sample(c(-1, 1), ncell(r), replace=TRUE)
    S <- setValues(r, s)
    Z <- 2*X + 3*Y + S
    return(Z)
}
```

We can call function `f` as many times as we like, below I use it four times. Note that the function has no arguments, but we still need to use the parenthesis `f()` to distinguish it from `f`, which is the function itself.

```
set.seed(777)
par(mfrow=c(2,2), mai=c(0.5,0.5,0.5,0.5))
for (i in 1:4) {
    pattern <- f()
    plot(pattern)
    text(pattern, cex=.75)
    contour(pattern, add=TRUE, labcex=1, lwd=2, col="red")
}
```



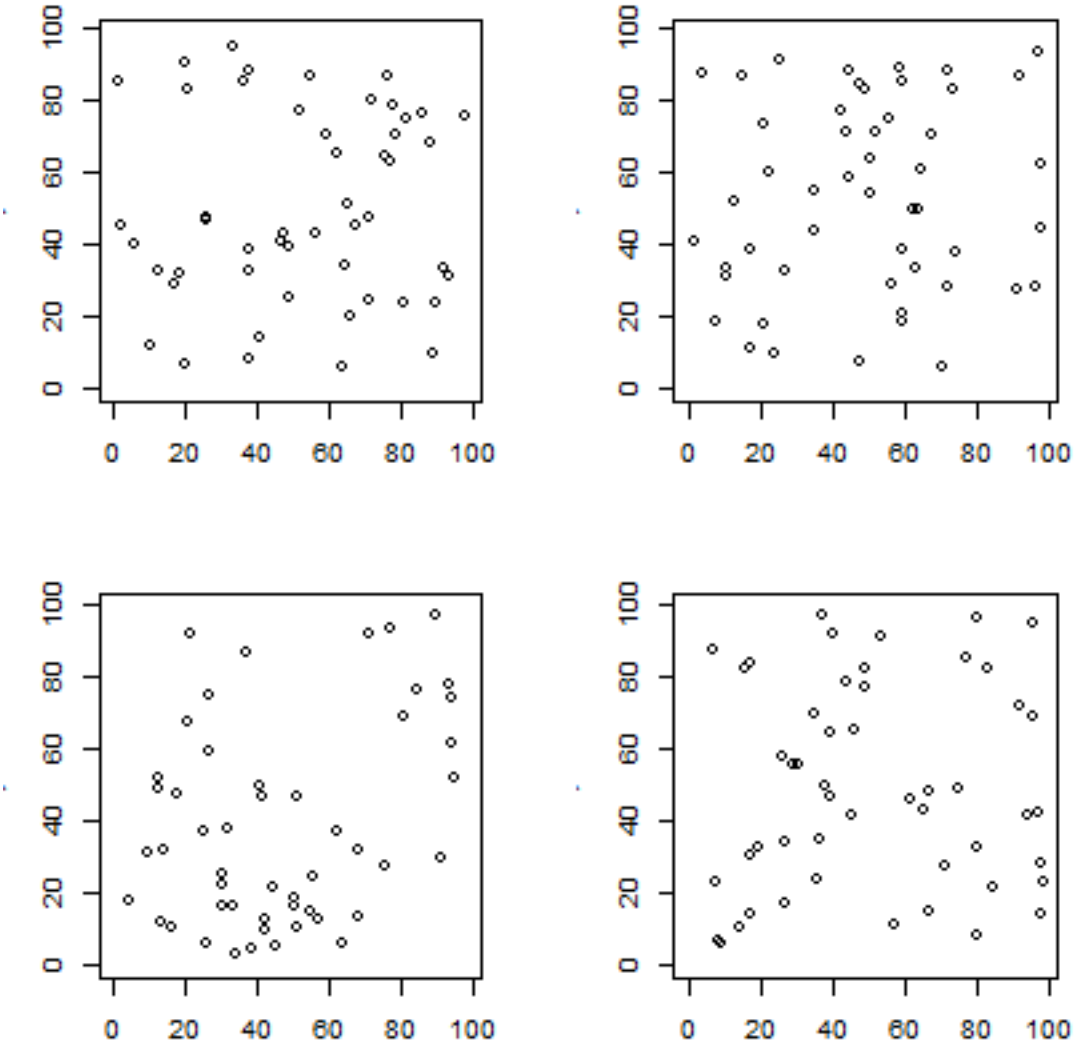
As you can see, there is variation between the four plots, but not much. The deterministic process has an overriding influence as the random component only adds or subtracts a value of 1.

So far we have created regular, gridded, patterns. Locations of "events" normally do not follow such a pattern (but they may be summarized that way). Here is how you can create simple dot maps of random events (following box "All the way: a chance map"; OSU page 98). I first create a function for a complete spatial random (CSR) process. Note the use of the `runif` (pronounced as "r unif" as it stands for "random uniform", there is also a `rnorm`, `rpois`, ...) function to create the x and y coordinates. For convenience, this function also plots the value. That is not typical, as in many cases you may want to create many random draws, but not plot them all. Therefore I added the logical argument "plot" (with default value `FALSE`) to the function.

```
csr <- function(n, r=99, plot=FALSE) {
    x <- runif(n, max=r)
    y <- runif(n, max=r)
    if (plot) {
        plot(x, y, xlim=c(0,r), ylim=c(0,r))
    }
}
```

Let's run the function four times; to create four realizations. Again, I use `set.seed` to assure that the maps are always the same "random" draws.

```
set.seed(0)
par(mfrow=c(2,2), mai=c(.5, .5, .5, .5))
for (i in 1:4) {
    csr(50, plot=TRUE)
}
```



4.2 Predicting patterns

I first show how you can recreate Table 4.1 with *R*. Note the use of function `choose` to get the "binomial coefficients" from this formula.

$$\frac{n!}{k!(n-k)!} = \binom{n}{k}$$

Everything else is just basic math.

```
events <- 0:10
combinations <- choose(10, events)
prob1 <- (1/8)^events
prob2 <- (7/8)^(10-events)
Pk <- combinations * prob1 * prob2
d <- data.frame(events, combinations, prob1, prob2, Pk)
round(d, 8)
##    events combinations      prob1     prob2         Pk
## 1       0            1 1.00000000 0.2630756 0.26307558
## 2       1           10 0.12500000 0.3006578 0.37582225
## 3       2           45 0.01562500 0.3436089 0.24160002
## 4       3          120 0.00195312 0.3926959 0.09203810
## 5       4          210 0.00024414 0.4487953 0.02300953
## 6       5          252 0.00003052 0.5129089 0.00394449
## 7       6          210 0.00000381 0.5861816 0.00046958
## 8       7          120 0.00000048 0.6699219 0.00003833
## 9       8           45 0.00000006 0.7656250 0.00000205
## 10      9           10 0.00000001 0.8750000 0.00000007
## 11     10            1 0.00000000 1.0000000 0.00000000
sum(d$Pk)
## [1] 1
```

Table 4.1 explains how value for the binomial distribution can be computed. As this is a "well-known" distribution (after all, it is the distribution you get when tossing a fair coin) there is a function to compute this directly.

```
b <- dbinom(0:10, 10, 1/8)
round(b, 8)
##  [1] 0.26307558 0.37582225 0.24160002 0.09203810 0.02300953 0.00394449
##  [7] 0.00046958 0.00003833 0.00000205 0.00000007 0.00000000
```

Similar functions exists for other commonly used distributions such as the uniform, normal, and Poisson distribution.

Now generate some quadrat counts and then compare the generated (observed) frequencies with the theoretical expectation. First the random points.

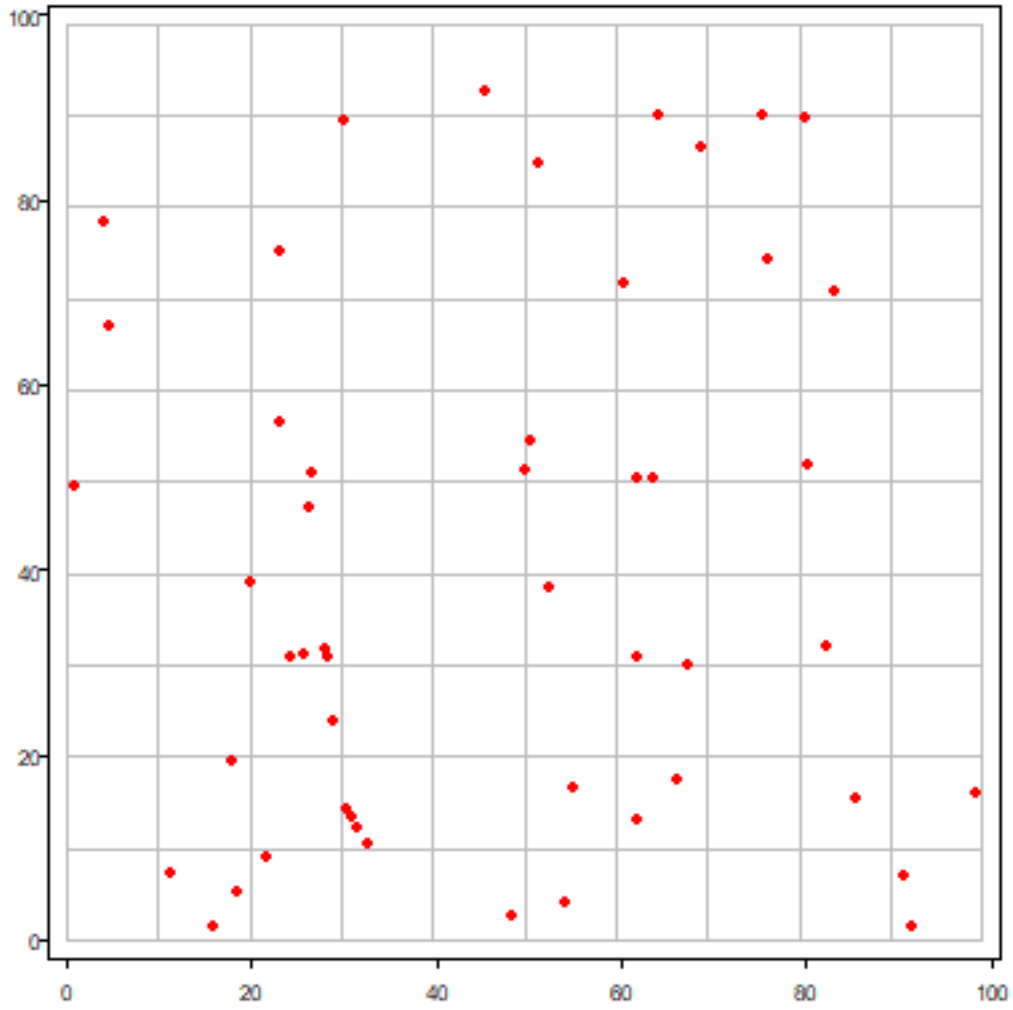
```
set.seed(1234)
x <- runif(50) * 99
y <- runif(50) * 99
```

And the quadrats.

```
r <- rast(xmin=0, xmax=99, ymin=0, ymax=99, ncol=10, nrow=10)
quads <- as.polygons(r)
```

And a plot to inspect them.

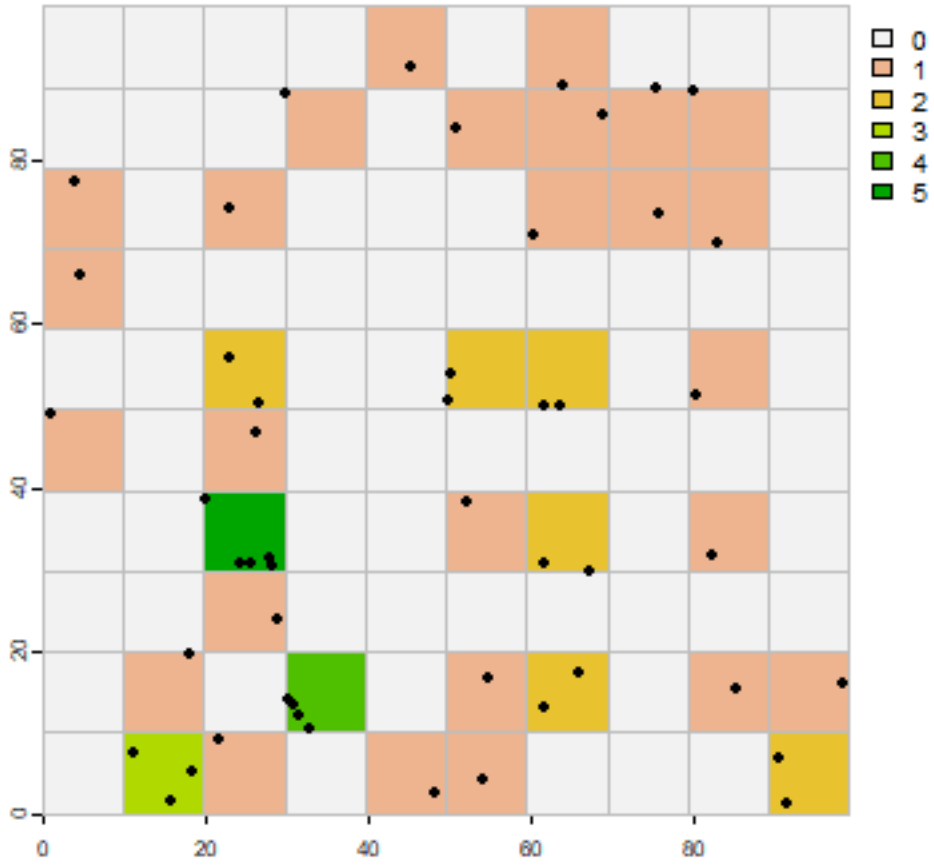
```
plot(quads, border="gray", pax=list(las=1))
points(x, y, col="red", pch=20)
```



A standard question to ask is whether it is likely that this pattern was generated by random process. We can do this by comparing the observed frequencies with the theoretically expected frequencies. Note that in a coin toss the probability of success is 1/2; here the probability of success (the random chance that a point lands in quadrat is 1/(number of quadrats).

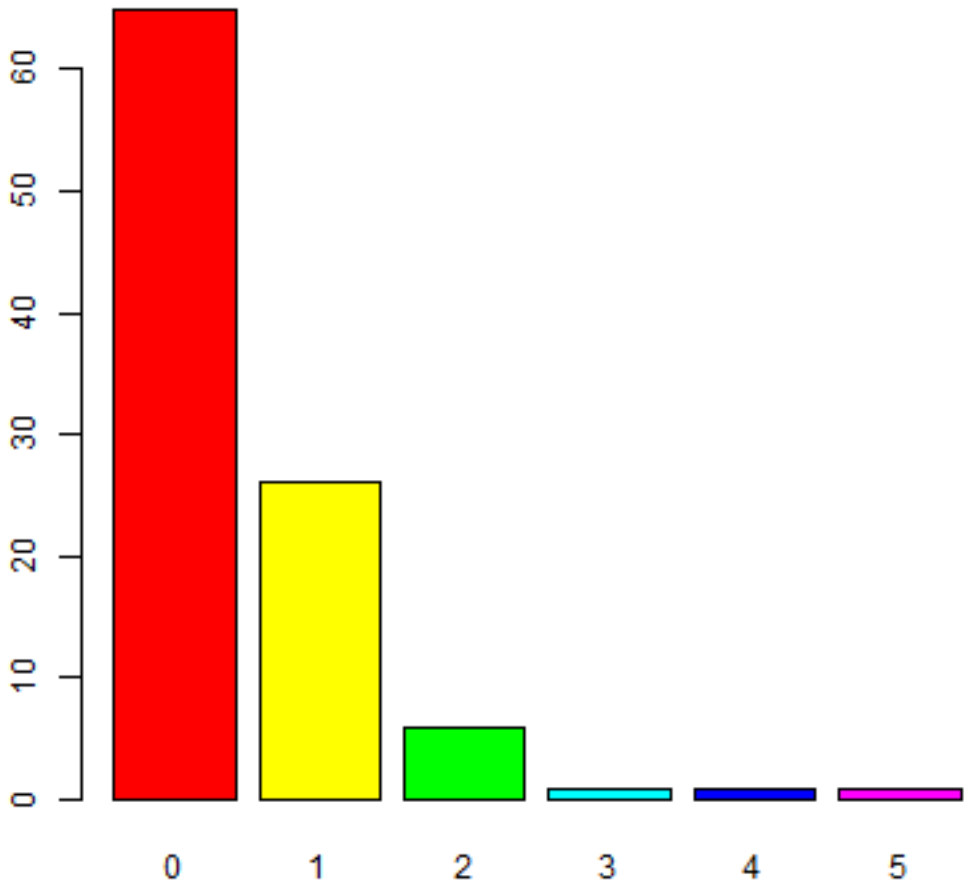
First count the number of points by quadrat (grid cell).

```
vxy <- vect(cbind(x,y))
vxy$v <- 1
p <- rasterize(vxy, r, "v", fun=length, background=0)

plot(p)
plot(quads, add=TRUE, border="gray")
points(x, y, pch=20)
```



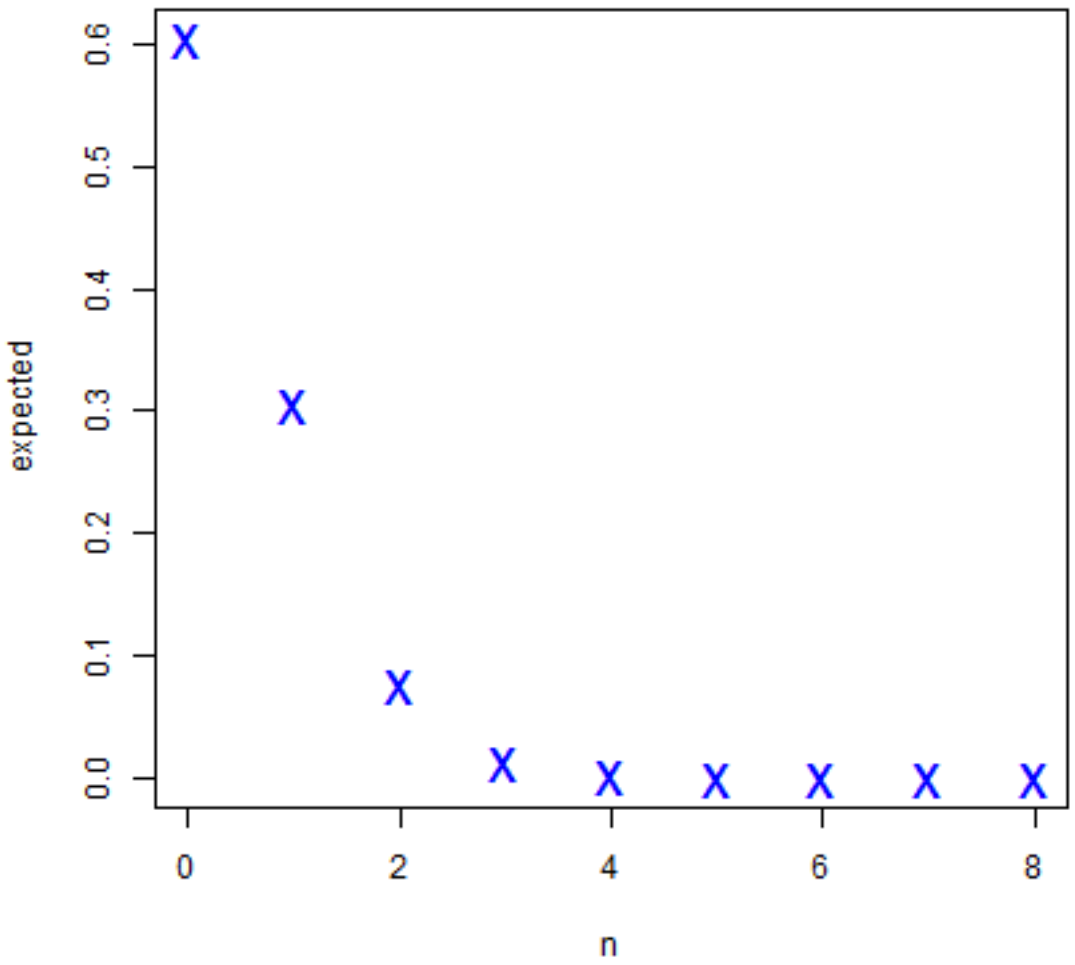
Get the frequency of the counts and make a barplot.

```
f <- freq(p)
f
##   layer value count
## 1     1     0    65
## 2     1     1    26
## 3     1     2     6
## 4     1     3     1
## 5     1     4     1
## 6     1     5     1
barplot(p)
```



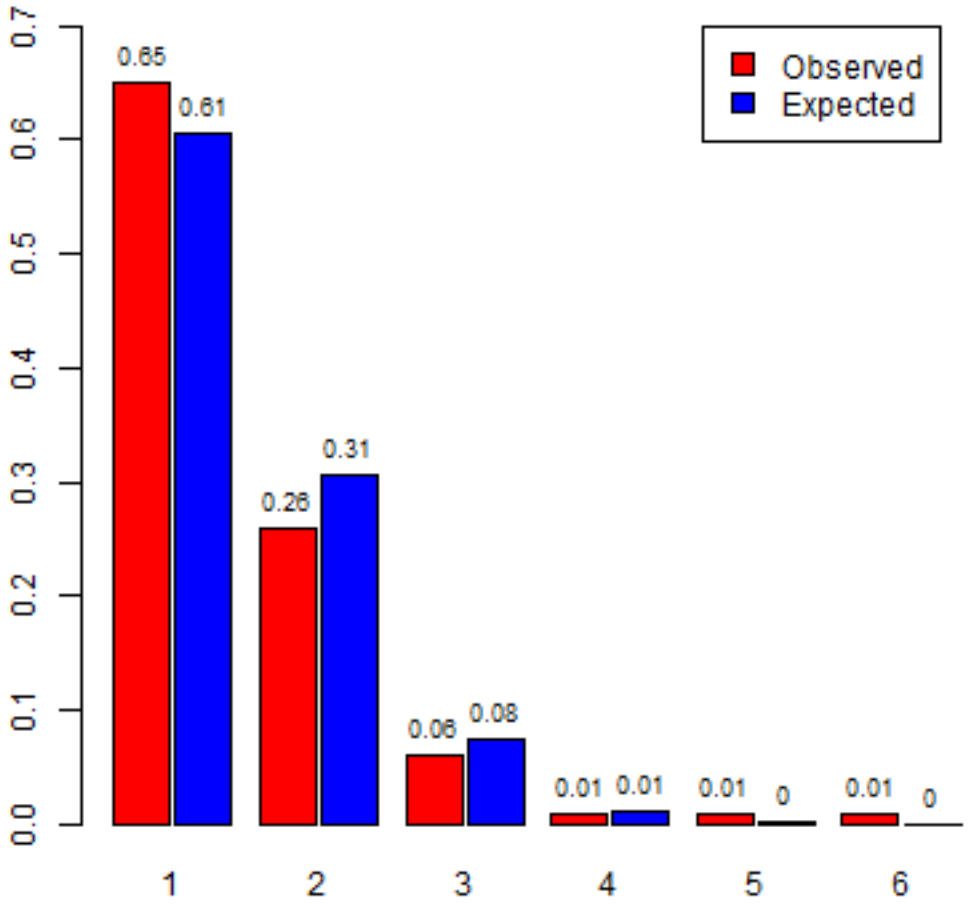
To compare these observed values to the expected frequencies from the binomial distribution we can use `expected <- dbinom(n, size, prob)`. In the book, this is `P(k, n, x)`.

```
n <- 0:8
prob <- 1 / ncell(r)
size <- 50
expected <- dbinom(n, size, prob)
round(expected, 5)
## [1] 0.60501 0.30556 0.07562 0.01222 0.00145 0.00013 0.00001 0.00000 0.00000
plot(n, expected, cex=2, pch="x", col="blue")
```



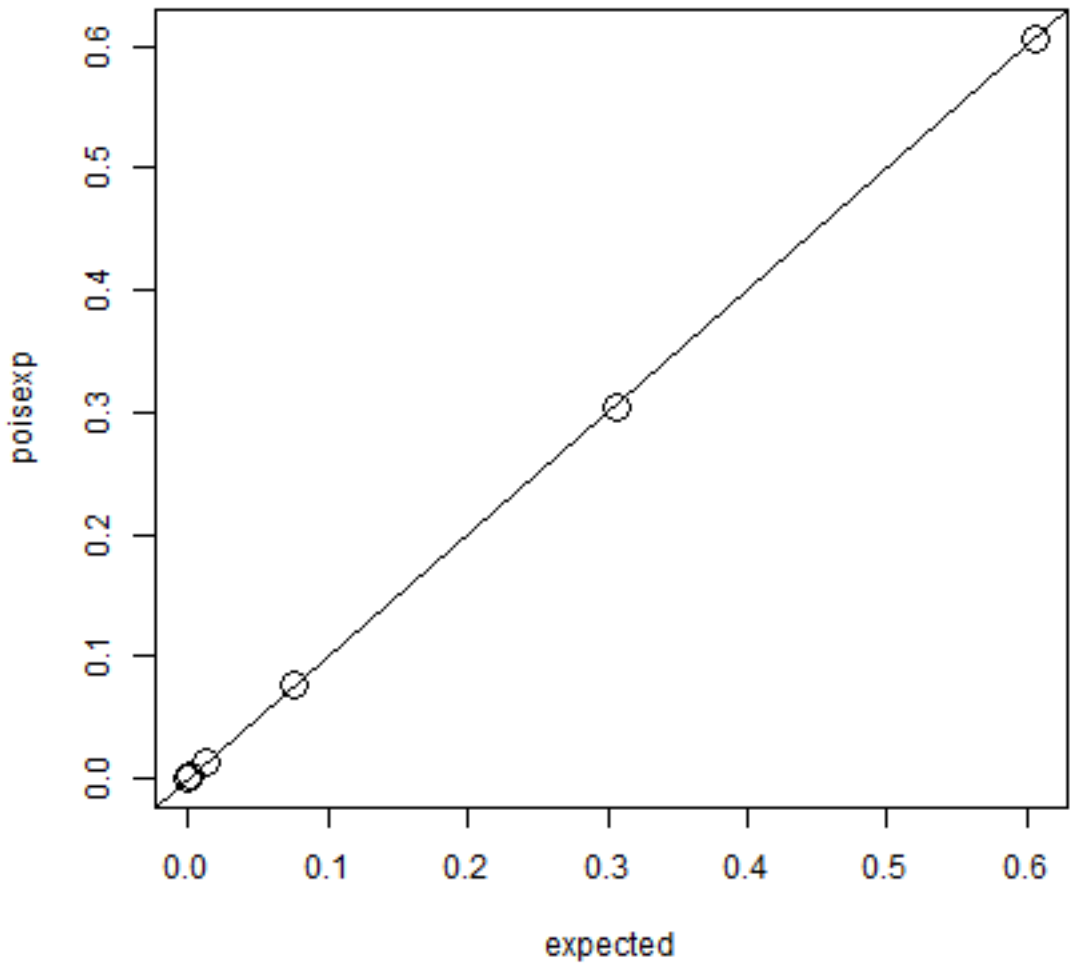
These numbers indicate that you would expect that most quadrats would have a point count of zero, a few would have 1 point, and very few more than that. Six or more points in a single cell is highly unlikely to happen if the data generating process in spatially random.

```
m <- rbind(f[,3]/100, expected[1:nrow(f)])
bp <- barplot(m, beside=T, names.arg =1:nrow(f), space=c(0.1, 0.5),
   ylim=c(0,0.7), col=c("red", "blue"))
text(bp, m, labels=round(m, 2), pos = 3, cex = .75)
legend(11, 0.7, c("Observed", "Expected"), fill=c("red", "blue"))
```



On page 106 it is discussed that the Poisson distribution can be a good approximation of the binomial distribution. Let's get the expected values for the Poisson distribution. The intensity λ (lambda) is the number of points divided by the number of quadrats.

```
poisexp <- dpois(0:8, lambda=50/100)
poisexp
## [1] 6.065307e-01 3.032653e-01 7.581633e-02 1.263606e-02 1.579507e-03
## [6] 1.579507e-04 1.316256e-05 9.401827e-07 5.876142e-08
plot(expected, poisexp, cex=2)
abline(0,1)
```



Pretty much the same, indeed.

4.3 Random Lines

See pp 110-111. Here is a function that draws random lines through a rectangle. It first takes a random point within the rectangle, and a random angle. Then it uses basic trigonometry to find the line segments (where the line intersects with the rectangle). It returns the line length, or the coordinates of the intersection and random point, for plotting.

```
randomLineInRectangle <- function(xmn=0, xmx=0.8, ymn=0, ymx=0.6, retXY=FALSE) {
    x <- runif(1, xmn, xmx)
    y <- runif(1, ymn, ymx)
    angle <- runif(1, 0, 359.99999999)
    if (angle == 0) {
        # vertical line, tan is infinite
        if (retXY) {
```

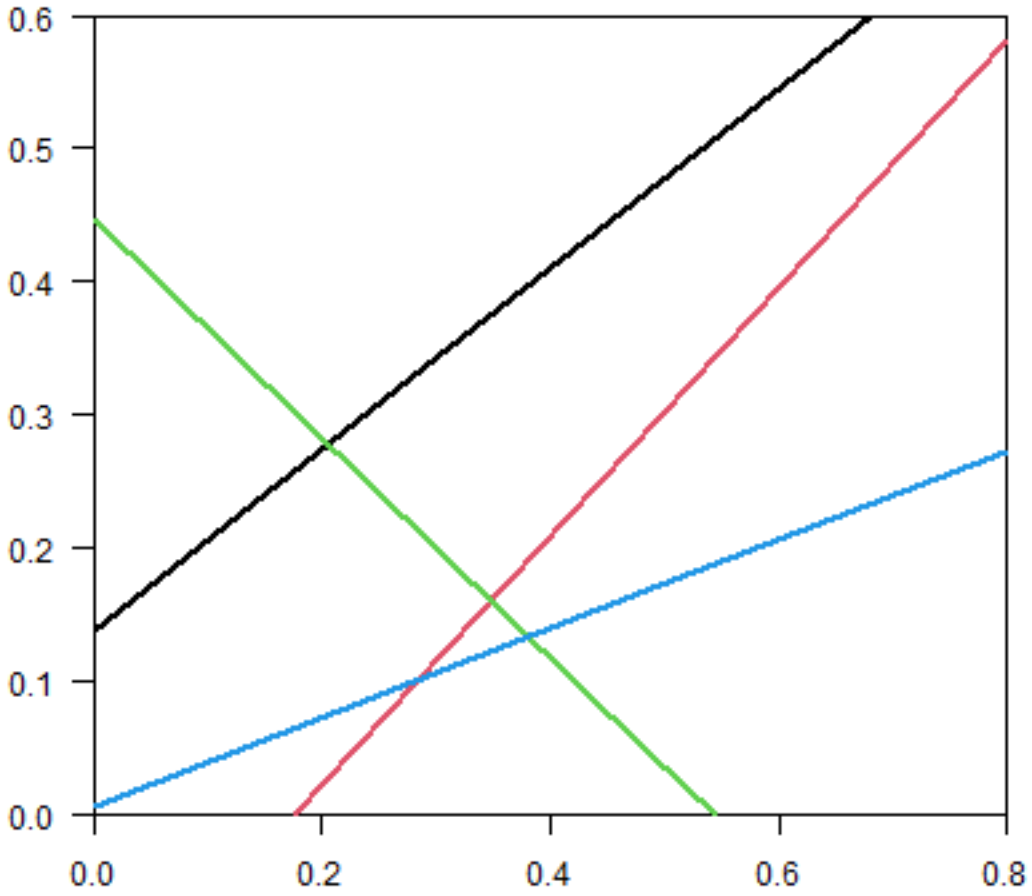
(continues on next page)

(continued from previous page)

```
            xy <- rbind(c(x, ymn), c(x, y), c(x, ymx))
            return(xy)
        }
        return(ymx - ymn)
    }
    tang <- tan(pi*angle/180)
    x1 <- max(xmn, min(xmx, x - y / tang))
    x2 <- max(xmn, min(xmx, x + (ymx-y) / tang))
    y1 <- max(ymn, min(ymx, y - (x-x1) * tang))
    y2 <- max(ymn, min(ymx, y + (x2-x) * tang))
    if (retXY) {
        xy <- rbind(c(x1, y1), c(x, y), c(x2, y2))
        return(xy)
    }
    sqrt((x2 - x1)^2 + (y2 - y1)^2)
}
```

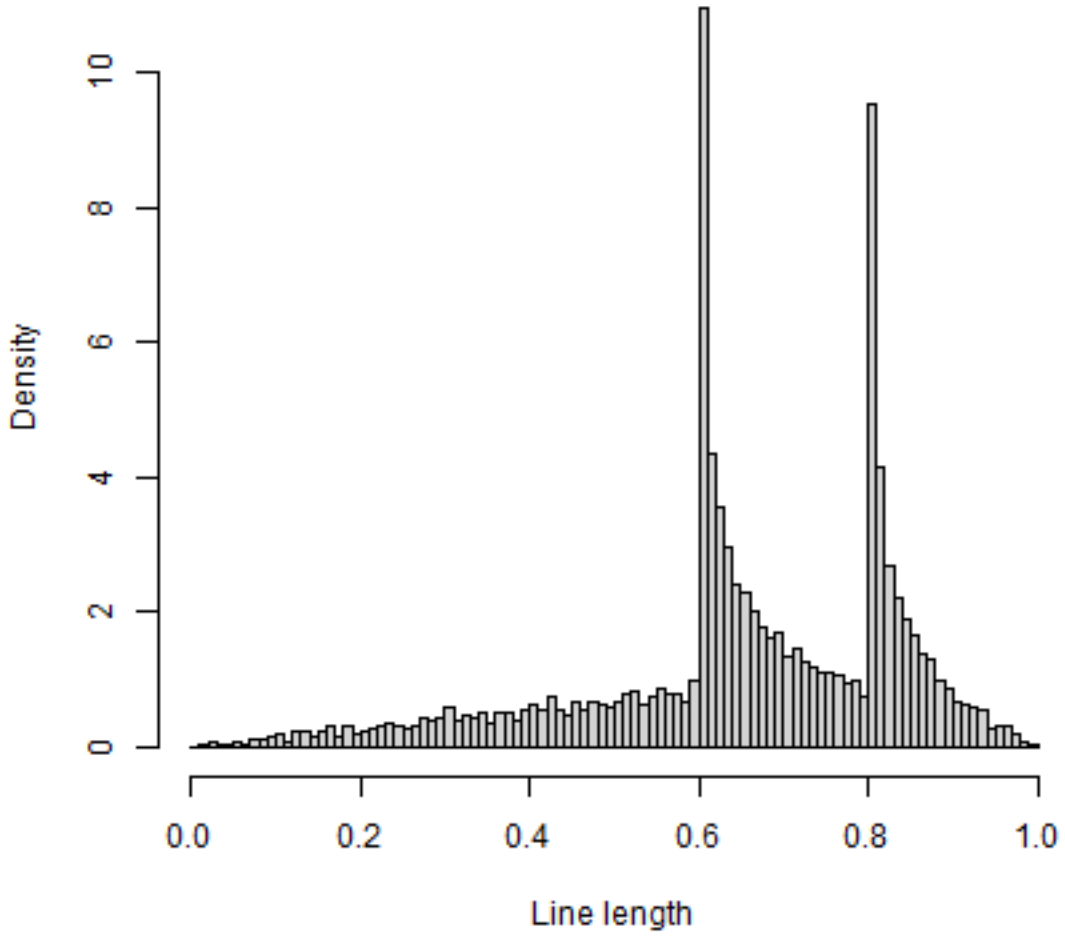
We can test it:

```
randomLineInRectangle()
## [1] 0.4953701
randomLineInRectangle()
## [1] 0.4470846

set.seed(999)
plot(NA, xlim=c(0, 0.8), ylim=c(0, 0.6), xaxs="i", yaxs="i", xlab="", ylab="", las=1)
for (i in 1:4) {
    xy <- randomLineInRectangle(retXY=TRUE)
    lines(xy, lwd=2, col=i)
    #points(xy, cex=2, pch=20, col=i)
}
```



And plot the density function

```
r <- replicate(10000, randomLineInRectangle())
hist(r, breaks=seq(0,1,0.01), xlab="Line length", main="", freq=FALSE)
```



Note that the density function is similar to Fig 4.6, but not the same (e.g. for values near zero).

4.4 Sitting comfortably?

See the box on page 110-111. There are four seats on the table. Let's number the seats 1, 2, 3, and 4. If two seats are occupied and the absolute difference between the seat numbers is 2, the customers sit straight across from each other. Otherwise, they sit across the corner from each other. What is the expected frequency of customers sitting straight across from each other? Let's simulate.

```
set.seed(0)
x <- replicate(10000, abs(diff(sample(1:4, 2))))
sum(x==2) / length(x)
## [1] 0.3408
```

That is one third, as expected given that this represents two of the six ways you can sit.

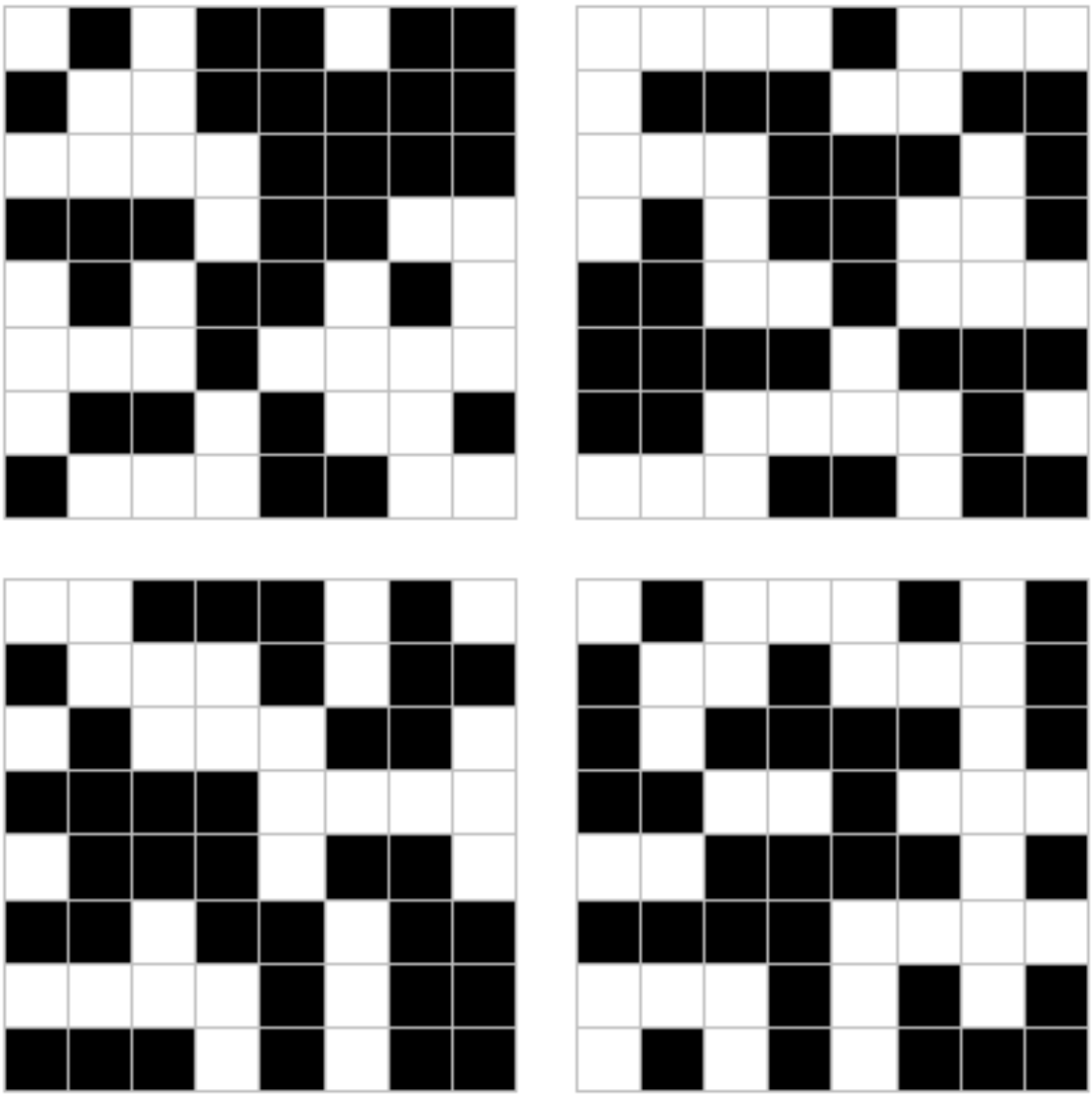
4.5 Random areas

Here is how you can create a function to create a "random chessboard" as described on page 114.

```
r <- rast(xmin=0, xmax=1, ymin=0, ymax=1, ncol=8, nrow=8)
p <- as.polygons(r)

chess <- function() {
    s <- sample(c(-1, 1), 64, replace=TRUE)
    values(r) <- s
    plot(r, col=c("black", "white"), legend=FALSE, axes=FALSE, mar=c(1,1,1,1))
    plot(p, add=T, border="gray")
}
```

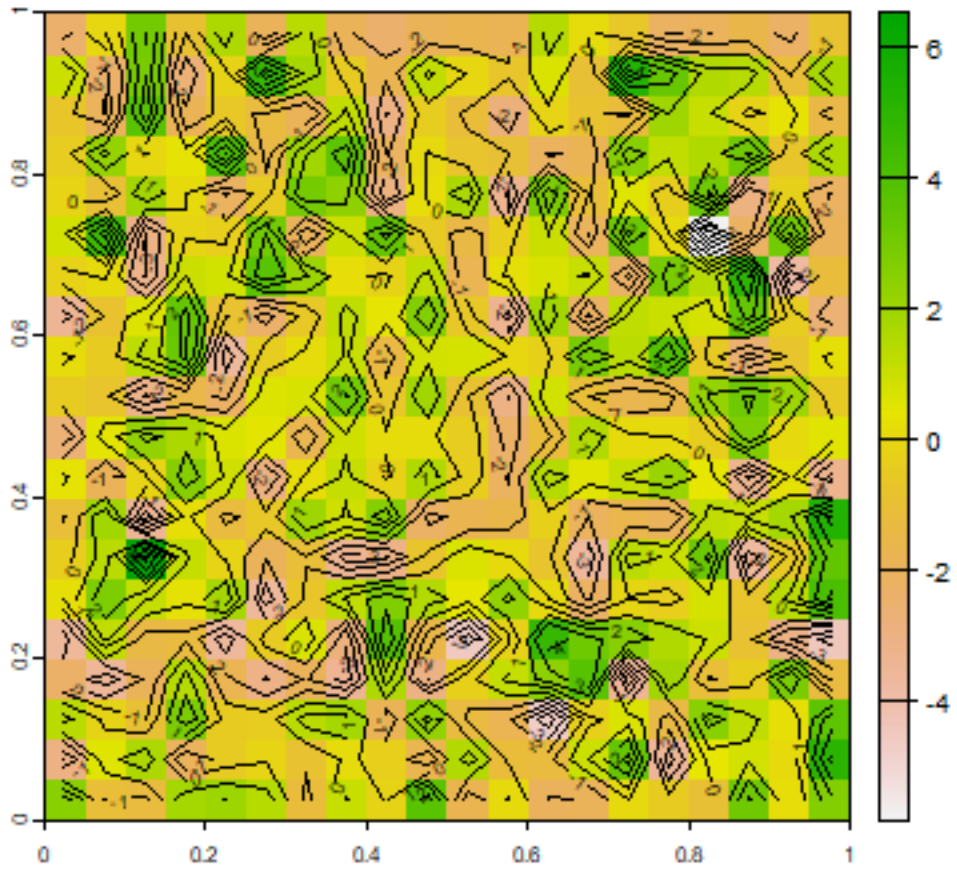
And create four realizations:

```
set.seed(0)
par(mfrow=c(2,2))
for (i in 1:4) {
    chess()
}
```



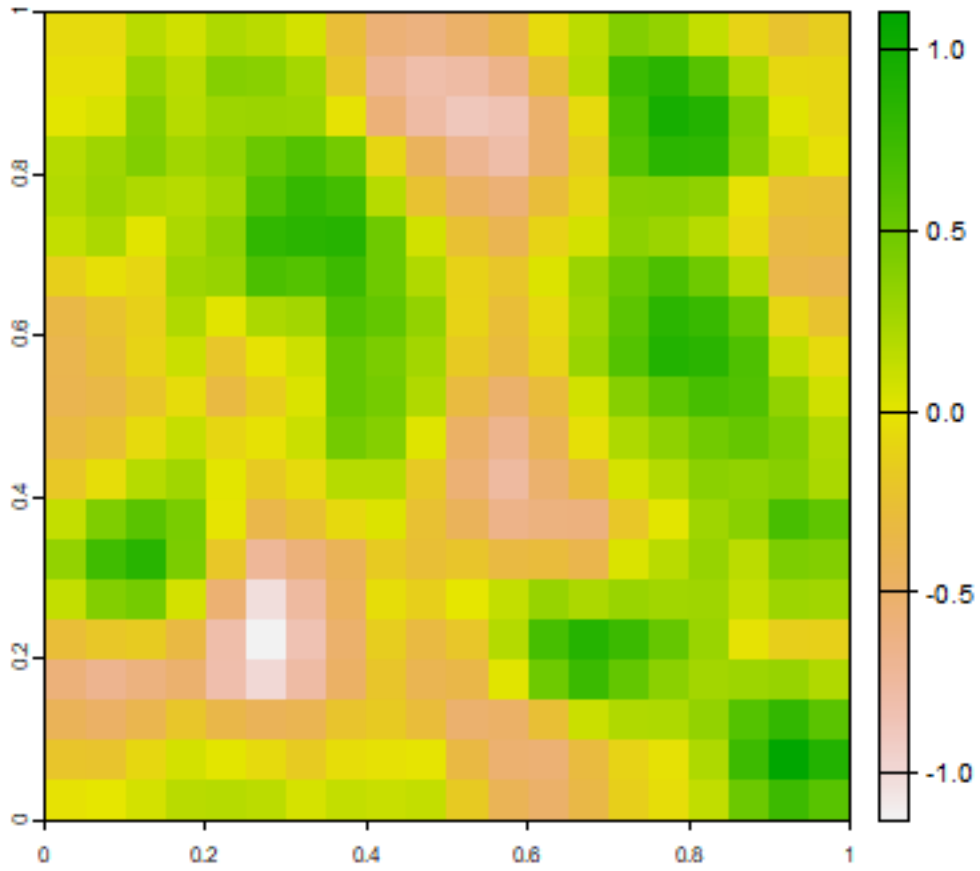
This is how you can create a random field (page 114/115)

```
r <- rast(xmin=0, xmax=1, ymin=0, ymax=1, ncol=20, nrow=20)
values(r) <- rnorm(ncell(r), 0, 2)
plot(r)
contour(r, add=T)
```



But a more realistic random field will have some spatial autocorrelation. We can create that with the focal function.

```
ra <- focal(r, w=matrix(1/9, nc=3, nr=3), na.rm=TRUE)
ra <- focal(ra, w=matrix(1/9, nc=3, nr=3), na.rm=TRUE)
plot(ra)
```



Questions

1. *Use the examples provided above to write a script that follows the "thought exercise to fix ideas" on page 98 of OSU. Use a function to generate the random numbers. Use vectorization, that is, avoid using a for -loop or if statements*
2. *Use the example of the CSR point distribution to write a script that uses a normal distribution, rather than a random uniform distribution (also see box "different distributions"; on page 99 of OSU) and compares the generated pattern to the expected values.*
3. *How could you, conceptually, statistically test whether the real chessboard used in games is generated by an independent random process? What raster processing function might you use? (feel free to attempt to implement this, perhaps in simplified form)*
4. *Can you explain the odd distribution pattern of random line lengths inside a rectangle? It can helpful to write some code to separate different cases*

CHAPTER

FIVE

POINT PATTERN ANALYSIS

5.1 Introduction

This page accompanies Chapter 5 of O'Sullivan and Unwin (2010).

We are using a dataset of crimes in a city. You can get these data from the `rspatial` package that you can install from github using the `remotes` package

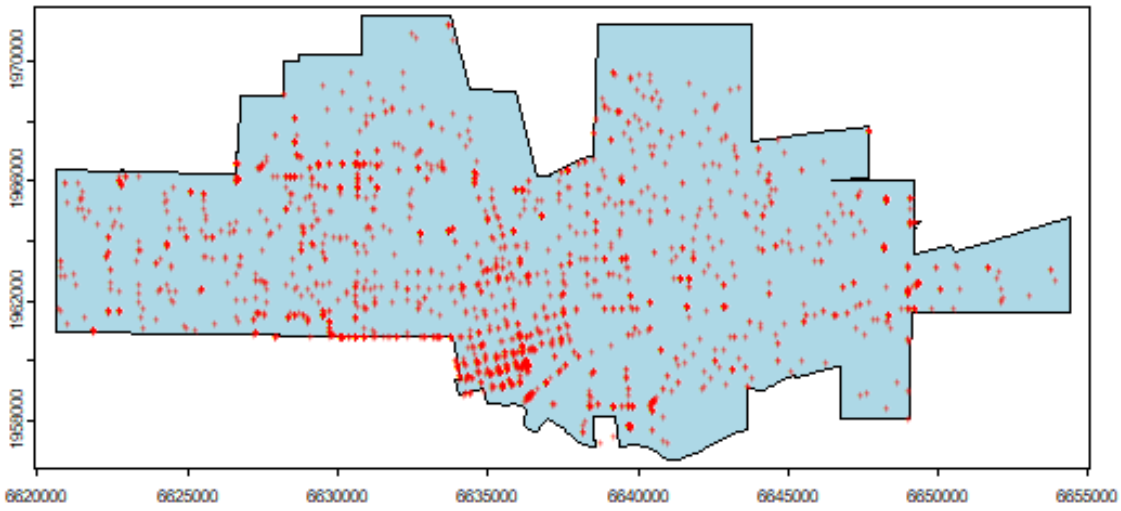
```
if (!require("rspat")) remotes::install_github('rspatial/rspat')
## Loading required package: rspat
## Loading required package: terra
## terra 1.7.62
```

Start by reading the data.

```
library(terra)
library(rspat)
city <- spat_data("city")
crime <- spat_data("crime")
```

Here is a map of both datasets.

```
par(mai=c(0,0,0,0))
plot(city, col='light blue')
points(crime, col='red', cex=.5, pch='+')
```



To find out what we are dealing with, we can make a sorted table of the incidence of crime types.

```
tb <- sort(table(crime$CATEGORY))[-1]
tb
##
##                Arson              Weapons              Robbery
##                    9                   15                   49
##           Auto Theft   Drugs or Narcotics  Commercial Burglary
##                   86                  134                  143
##          Grand Theft             Assaults                  DUI
##                  143                  172                  212
## Residential Burglary     Vehicle Burglary      Drunk in Public
##                  219                  221                  232
##             Vandalism          Petty Theft
##                  355                  665
```

Let's get the coordinates of the crime data, and for this exercise, remove duplicate crime locations. These are the "events" we will use below (later we'll go back to the full data set).

```
xy <- geom(crime)[, c("x", "y")]
dim(xy)
## [1] 2661    2
xy <- unique(xy)
dim(xy)
## [1] 1208    2
head(xy)
##            x       y
## [1,] 6628868 1963718
## [2,] 6632796 1964362
## [3,] 6636855 1964873
## [4,] 6626493 1964343
## [5,] 6639506 1966094
## [6,] 6640478 1961983
```

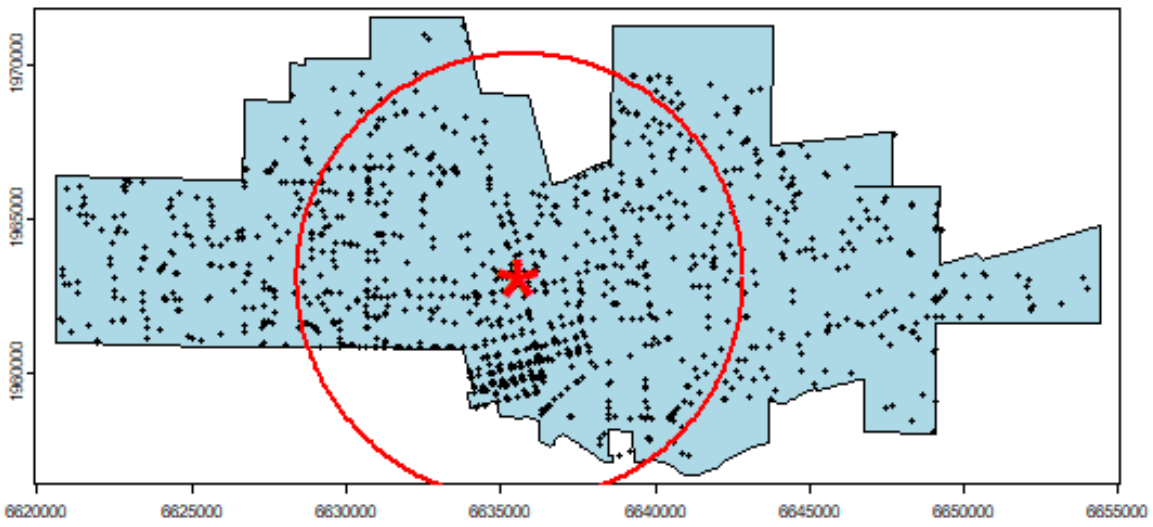
5.2 Basic statistics

Compute the mean center and standard distance for the crime data (see page 125 of OSU).

```
# mean center
mc <- apply(xy, 2, mean)
# standard distance
sd <- sqrt(sum((xy[,1] - mc[1])^2 + (xy[,2] - mc[2])^2) / nrow(xy))
```

Plot the data to see what we've got. I add a summary circle (as in Fig 5.2) by dividing the circle in 360 points and compute bearing in radians. I do not think this is particularly helpful, but it might be in other cases. And it is always fun to figure out how to do tis.

```
plot(city, col='light blue')
points(crime, cex=.5)
points(cbind(mc[1], mc[2]), pch='*', col='red', cex=5)

# make a circle
bearing <- 1:360 * pi/180
cx <- mc[1] + sd * cos(bearing)
cy <- mc[2] + sd * sin(bearing)
circle <- cbind(cx, cy)
lines(circle, col='red', lwd=2)
```



5.3 Density

Here is a basic approach to computing point density.

```
CityArea <- expanse(city)
dens <- nrow(xy) / CityArea
```

Question 1a:*What is the unit of 'dens'?*

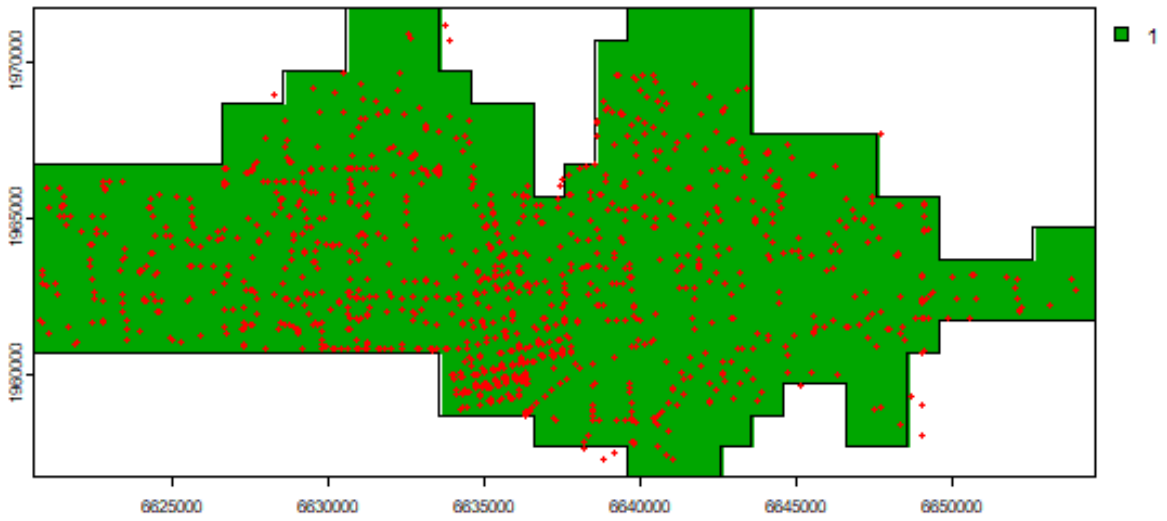
Question 1b:*What is the number of crimes per km^2?*

To compute quadrat counts (as on p.127-130), I first create quadrats (a SpatRaster). I get the extent for the raster from the city polygon, and then assign an an arbitrary resolution of 1000. (In real life one should always try a range of resolutions, I think).

```
r <- rast(city, res=1000)
r
## class       : SpatRaster
## dimensions  : 15, 34, 1  (nrow, ncol, nlyr)
## resolution  : 1000, 1000  (x, y)
## extent      : 6620591, 6654591, 1956730, 1971730  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=lcc +lat_0=37.6666666666667 +lon_0=-122 +lat_1=38.3333333333333␣
↪+lat_2=39.8333333333333 +x_0=2000000 +y_0=500000.0000000001 +datum=WGS84 +units=us-ft␣
↪+no_defs
```

To find the cells that are in the city, and for easy display, I create polygons from the SpatRaster.

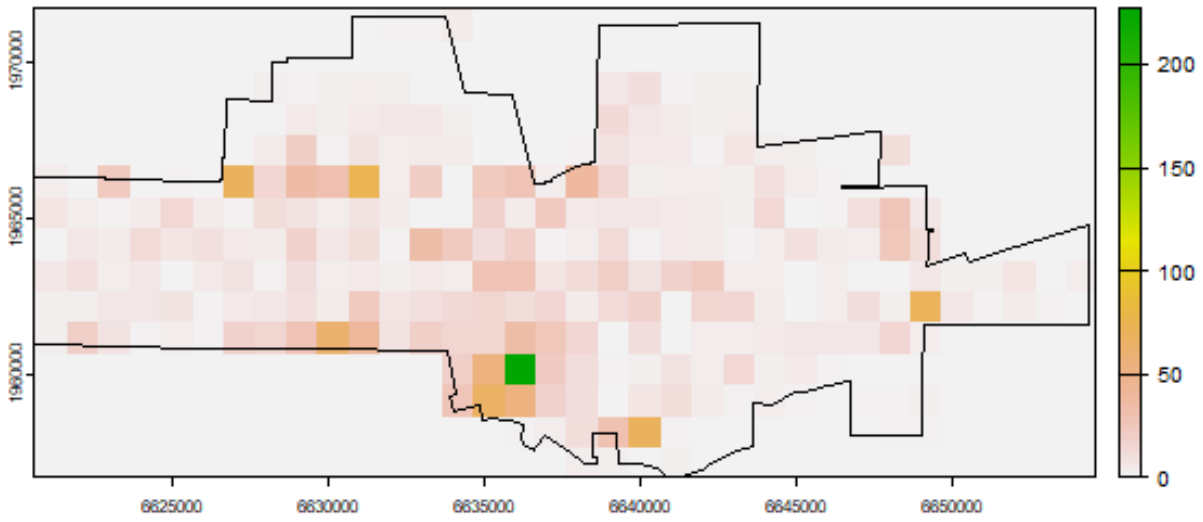
```
r <- rasterize(city, r)
plot(r)
quads <- as.polygons(r)
plot(quads, add=TRUE)
points(crime, col='red', cex=.5)
```



The number of events in each quadrat can be counted using the 'rasterize' function. That function can be used to summarize the number of points within each cell, but also to compute statistics based on the 'marks' (attributes). For

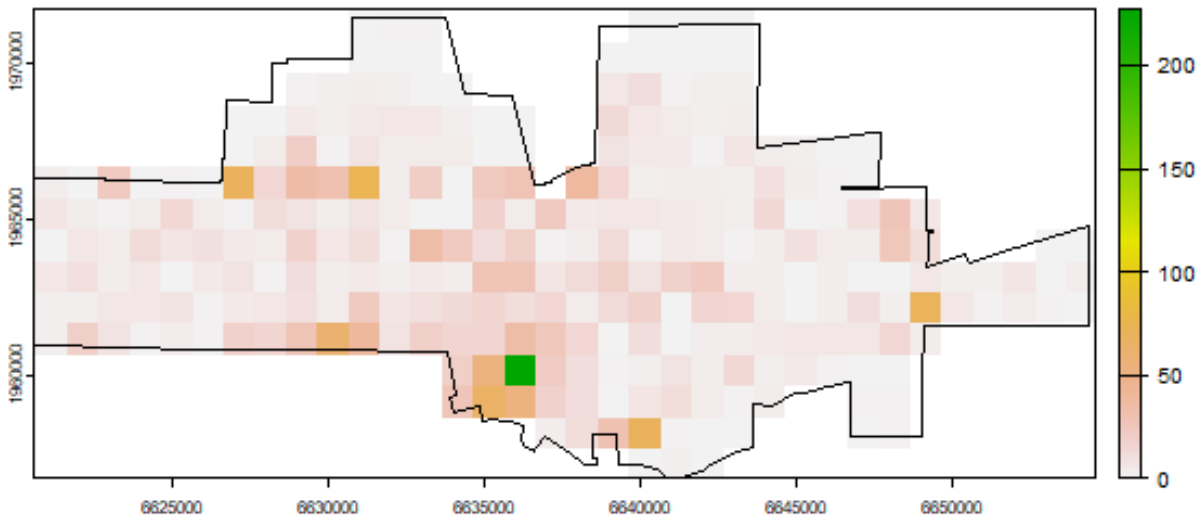
example we could compute the number of different crime types) by changing the 'fun' argument to another function (see ?rasterize).

```
nc <- rasterize(crime, r, fun=length, background=0)
plot(nc)
plot(city, add=TRUE)
```



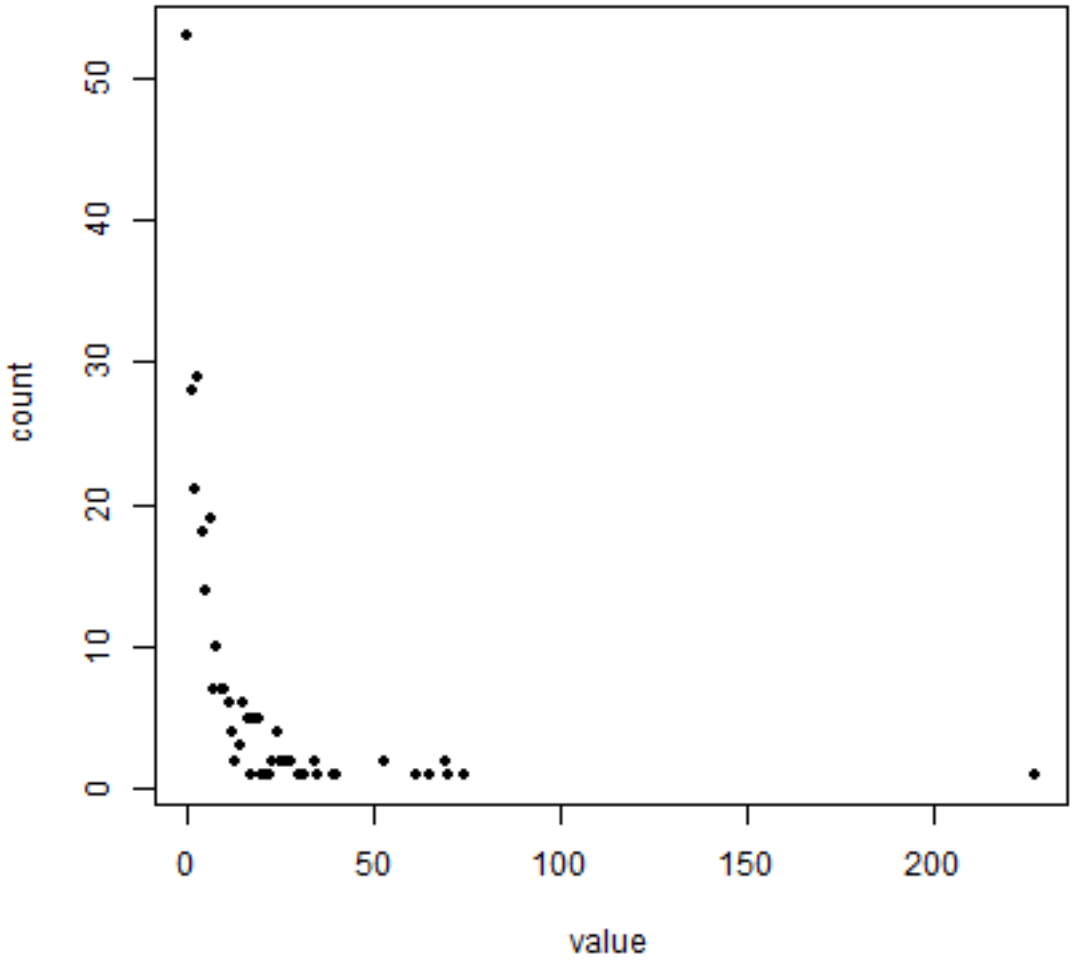
nc has crime counts. As we only have data for the city, the areas outside of the city need to be excluded. We can do that with the mask function (see ?mask).

```
ncrimes <- mask(nc, r)
plot(ncrimes)
plot(city, add=TRUE)
```



That looks better. Now let's get the frequencies.

```
f <- freq(ncrimes)
head(f)
##   layer value count
## 1     1     0    53
## 2     1     1    28
## 3     1     2    21
## 4     1     3    29
## 5     1     4    18
## 6     1     5    14
f <- f[,-1]
plot(f, pch=20)
```



Does this look like a pattern you would have expected?

Compute the average number of cases per quadrat.

```
# number of quadrats
quadrats <- sum(f[,2])
# number of cases
cases <- sum(f[,1] * f[,2])
mu <- cases / quadrats
mu
## [1] 9.293286
```

And create a table like Table 5.1 on page 130

```
ff <- data.frame(f)
colnames(ff) <- c('K', 'X')
ff$Kmu <- ff$K - mu
ff$Kmu2 <- ff$Kmu^2
ff$XKmu2 <- ff$Kmu2 * ff$X
head(ff)
##   K  X       Kmu     Kmu2     XKmu2
## 1 0 53 -9.293286 86.36517 4577.3539
## 2 1 28 -8.293286 68.77860 1925.8007
## 3 2 21 -7.293286 53.19202 1117.0325
## 4 3 29 -6.293286 39.60545 1148.5581
## 5 4 18 -5.293286 28.01888  504.3398
## 6 5 14 -4.293286 18.43231  258.0523
```

The observed variance s^2 is

```
s2 <- sum(ff$XKmu2) / (sum(ff$X)-1)
s2
## [1] 325.8889
```

And the VMR is

```
VMR <- s2 / mu
VMR
## [1] 35.06713
```

Question 2:*What does this VMR score tell us about the point pattern?*

5.4 Distance based measures

As we are using a *planar coordinate system* we can use the dist function to compute the distances between pairs of points. Contrary to what the books says, if we were using longitude/latitude we could compute distance via spherical trigonometry functions. These are available in the sp, raster, and notably the geosphere package (among others). For example, see `raster::distance`.

```
d <- dist(xy)
class(d)
## [1] "dist"
```

I want to coerce the dist object to a matrix, and ignore distances from each point to itself (the zeros on the diagonal).

```
dm <- as.matrix(d)
dm[1:5, 1:5]
##           1        2         3         4         5
## 1     0.000 3980.843  8070.429  2455.809 10900.016
## 2  3980.843    0.000  4090.992  6303.450  6929.439
## 3  8070.429 4090.992     0.000 10375.958  2918.349
## 4  2455.809 6303.450 10375.958     0.000 13130.236
## 5 10900.016 6929.439  2918.349 13130.236     0.000
diag(dm) <- NA
dm[1:5, 1:5]
##           1        2         3         4         5
## 1        NA 3980.843  8070.429  2455.809 10900.016
## 2  3980.843       NA  4090.992  6303.450  6929.439
## 3  8070.429 4090.992        NA 10375.958  2918.349
## 4  2455.809 6303.450 10375.958        NA 13130.236
## 5 10900.016 6929.439  2918.349 13130.236        NA
```

To get, for each point, the minimum distance to another event, we can use the 'apply' function. Think of the rows as each point, and the columns of all other points (vice versa could also work).

```
dmin <- apply(dm, 1, min, na.rm=TRUE)
head(dmin)
##         1         2         3         4         5         6
## 266.07892 293.58874  47.90260 140.80688  40.06865 510.41231
```

Now it is trivial to get the mean nearest neighbour distance according to formula 5.5, page 131.

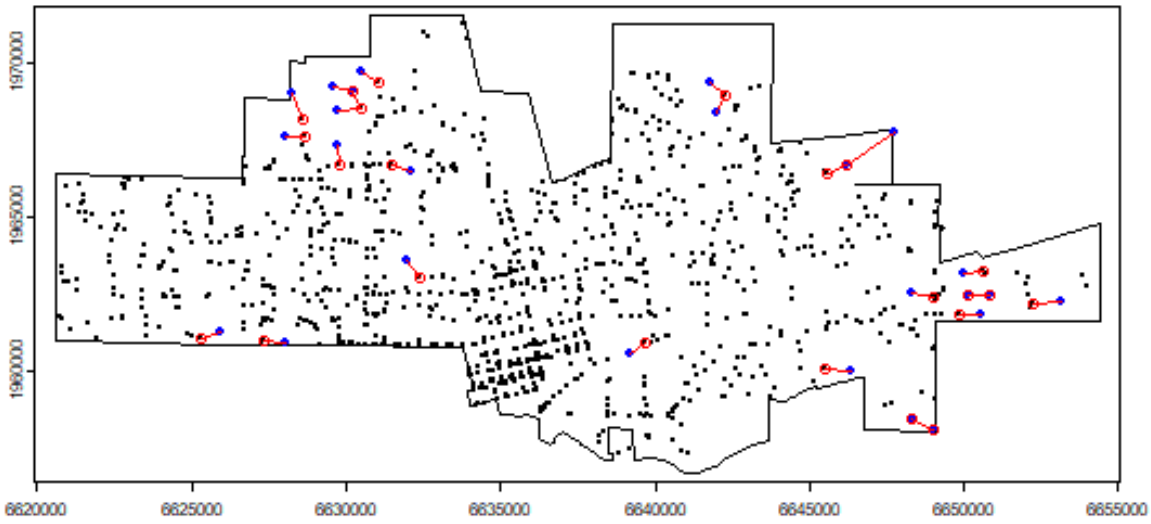
```
mdmin <- mean(dmin)
```

Do you want to know, for each point, *Which* point is its nearest neighbour? Use the 'which.min' function (but note that this ignores the possibility of multiple points at the same minimum distance).

```
wdmin <- apply(dm, 1, which.min)
```

And what are the most isolated cases? That is, which cases are the furthest away from their nearest neighbor. Below I plot the top 25 cases. It is a bit complicated.

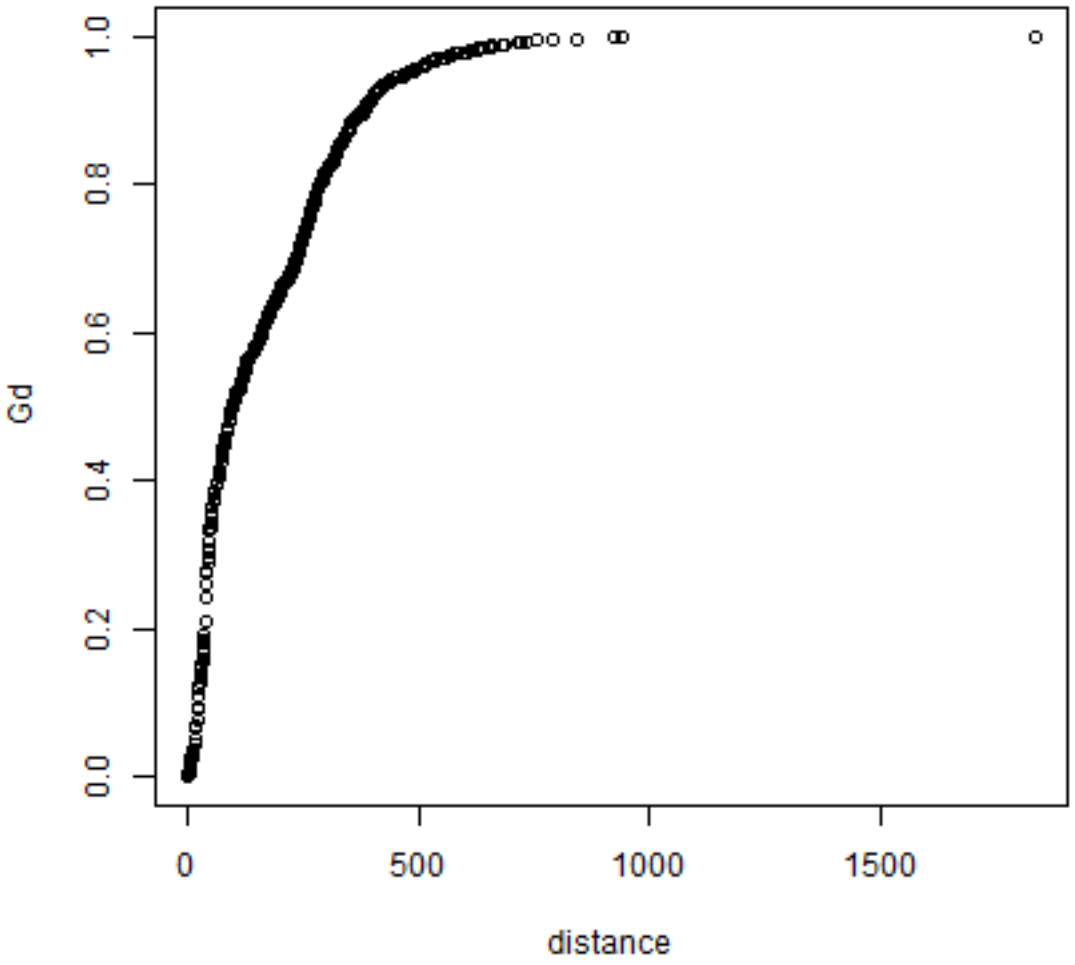
```
plot(city)
points(crime, cex=.1)
ord <- rev(order(dmin))

far25 <- ord[1:25]
neighbors <- wdmin[far25]

points(xy[far25, ], col='blue', pch=20)
points(xy[neighbors, ], col='red')

# drawing the lines, easiest via a loop
for (i in far25) {
    lines(rbind(xy[i, ], xy[wdmin[i], ]), col='red')
}
```



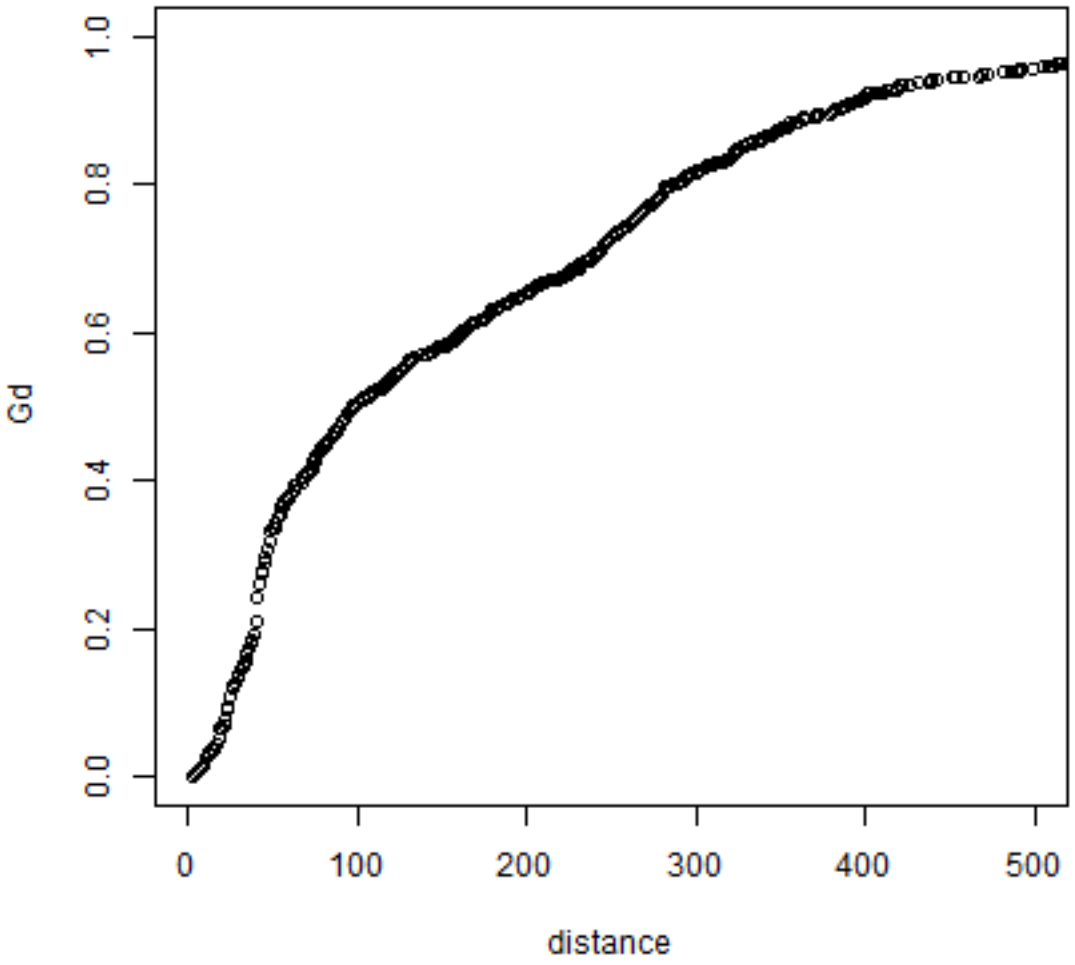
Note that some points, but actually not that many, are both isolated and a neighbor to another isolated point.

On to the *G* function.

```
max(dmin)
## [1] 1829.738
# get the unique distances (for the x-axis)
distance <- sort(unique(round(dmin)))
# compute how many cases there with distances smaller that each x
Gd <- sapply(distance, function(x) sum(dmin < x))
# normalize to get values between 0 and 1
Gd <- Gd / length(dmin)
plot(distance, Gd)
```




```
# using xlim to exclude the extremes
plot(distance, Gd, xlim=c(0,500))
```

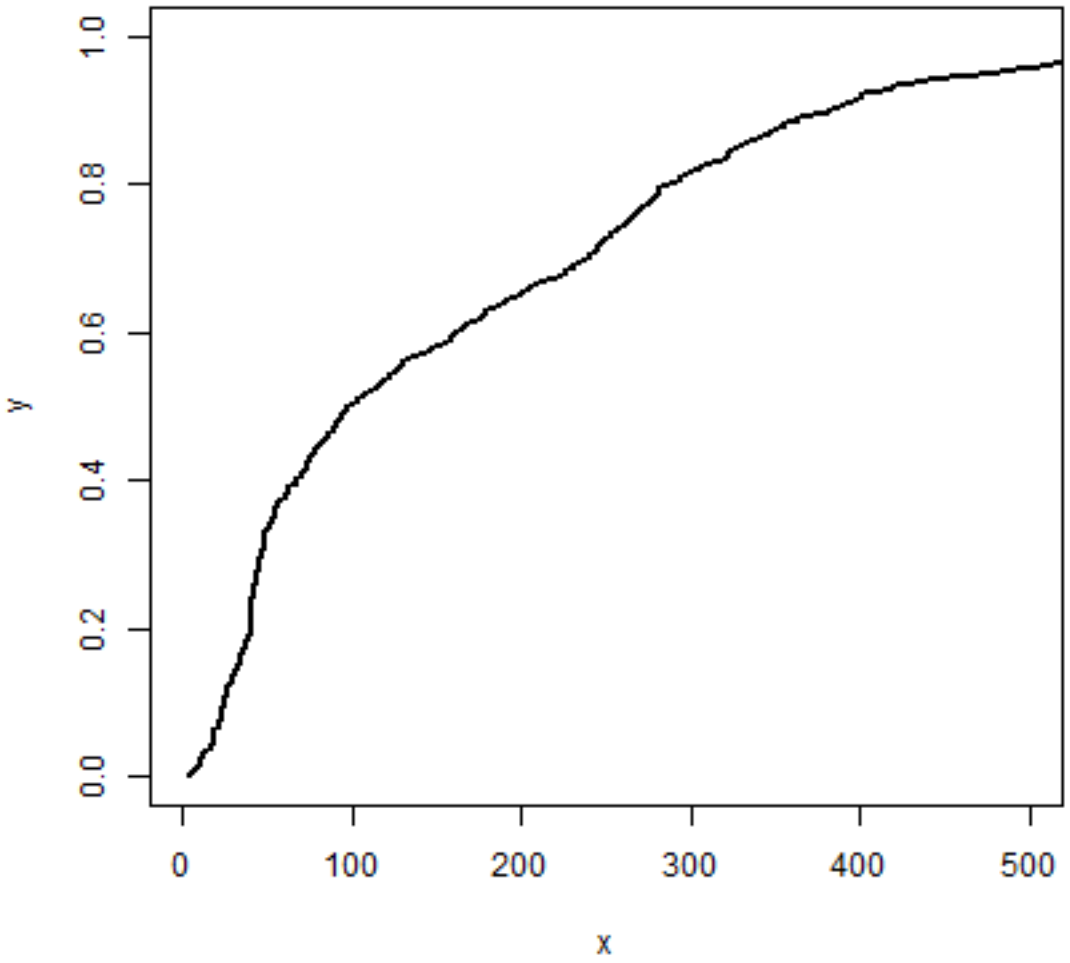


Here is a function to show these values in a more standard way.

```
stepplot <- function(x, y, type='l', add=FALSE, ...) {
    x <- as.vector(t(cbind(x, c(x[-1], x[length(x)]))))
    y <- as.vector(t(cbind(y, y)))
  if (add) {
     lines(x,y, ...)
  } else {
       plot(x,y, type=type, ...)
  }
}
```

And use it for our G function data.

```
stepplot(distance, Gd, type='l', lwd=2, xlim=c(0,500))
```



The steps are so small in our data, that you hardly see the difference.

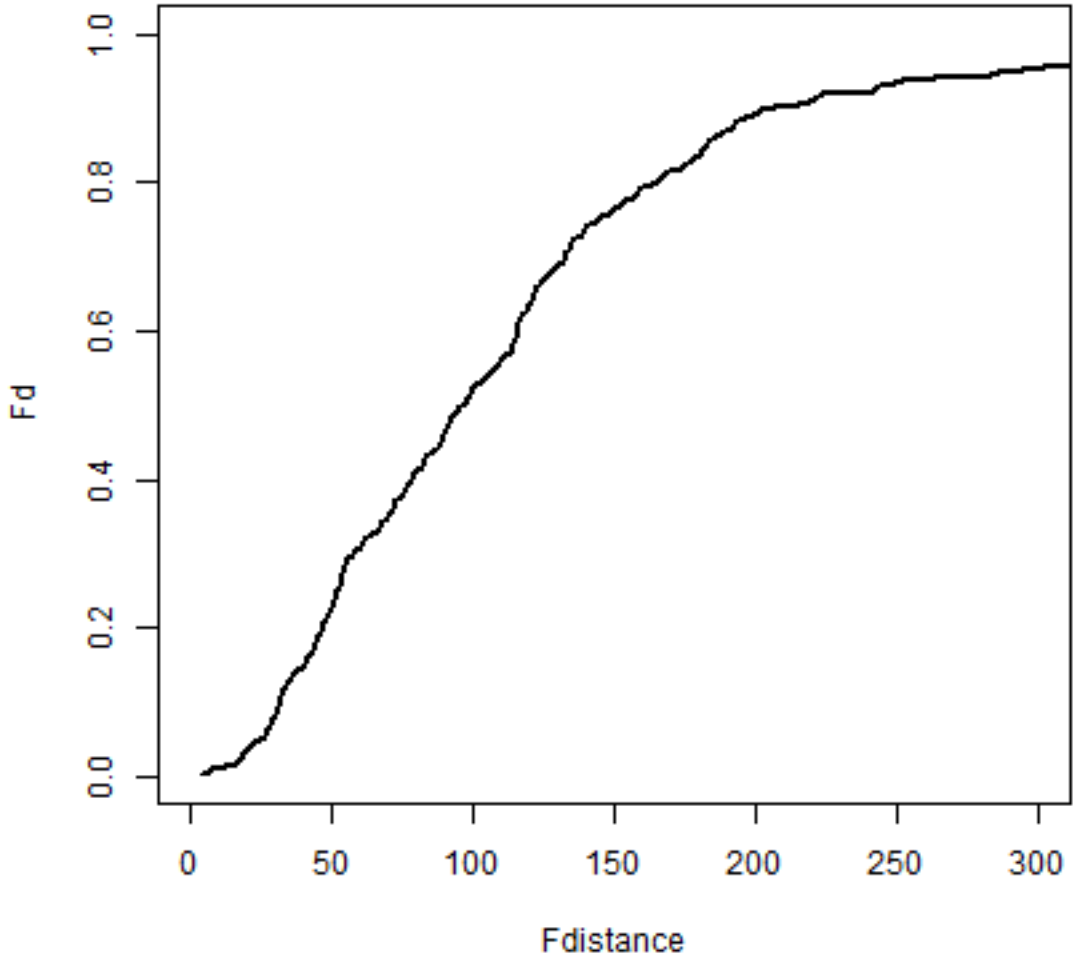
I use the centers of previously defined raster cells to compute the F function.

```
# get the centers of the 'quadrats' (raster cells)
p <- as.points(r)
v <- vect(xy, crs=crs(p))
# compute distance from all crime sites to these cell centers
d2 <- distance(p, v)

# the remainder is similar to the G function
Fdistance <- sort(unique(round(d2)))
mind <- apply(d2, 1, min)
Fd <- sapply(Fdistance, function(x) sum(mind < x))
Fd <- Fd / length(mind)
```

(continues on next page)

(continued from previous page)

```
plot(Fdistance, Fd, type='l', lwd=2, xlim=c(0,300))
```



Compute the expected distribution (5.12 on page 145).

```
ef <- function(d, lambda) {
  E <- 1 - exp(-1 * lambda * pi * d^2)
}
expected <- ef(0:2000, dens)
```

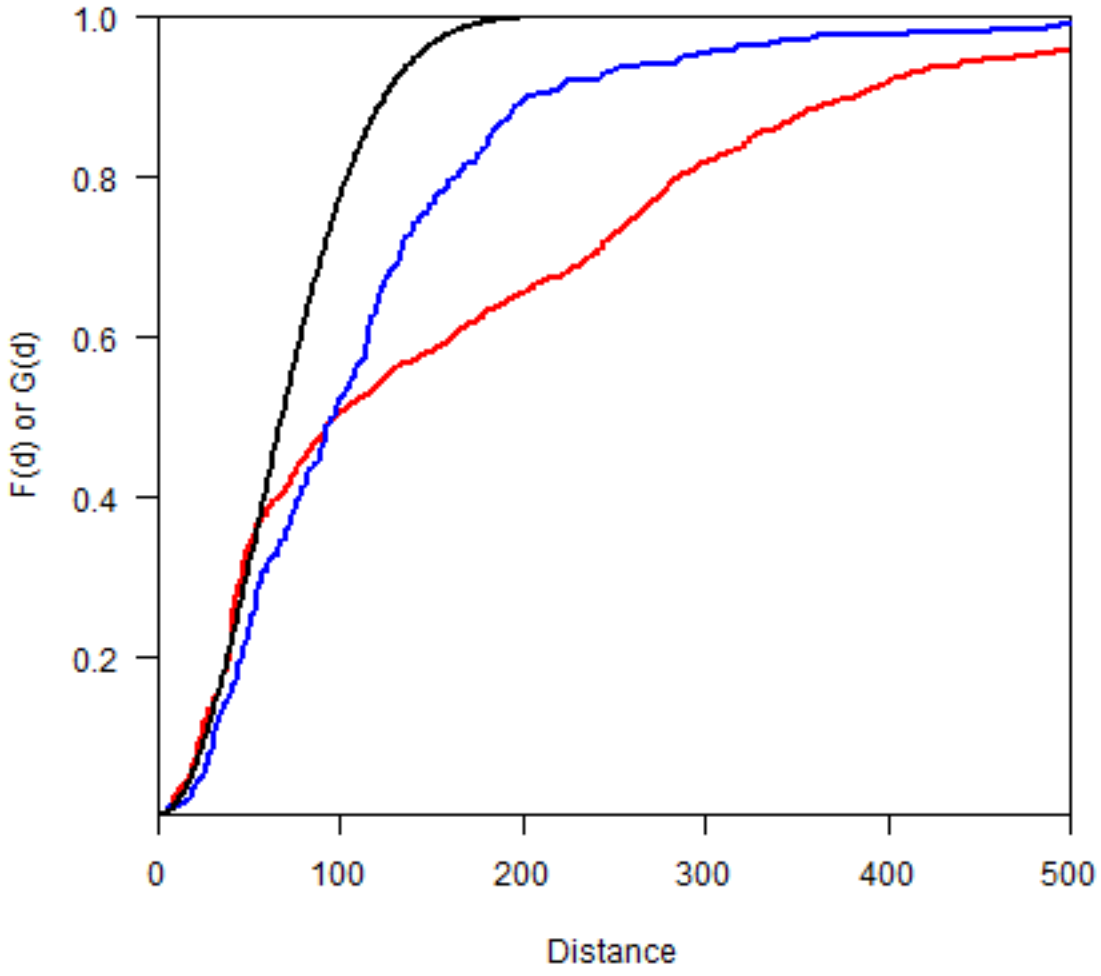
We can combine F and G on one plot.

```
plot(distance, Gd, type='l', lwd=2, col='red', las=1, xlim=c(0,500),
    ylab='F(d) or G(d)', xlab='Distance', yaxs="i", xaxs="i")
lines(Fdistance, Fd, lwd=2, col='blue')
lines(0:2000, expected, lwd=2)
```

(continues on next page)

(continued from previous page)

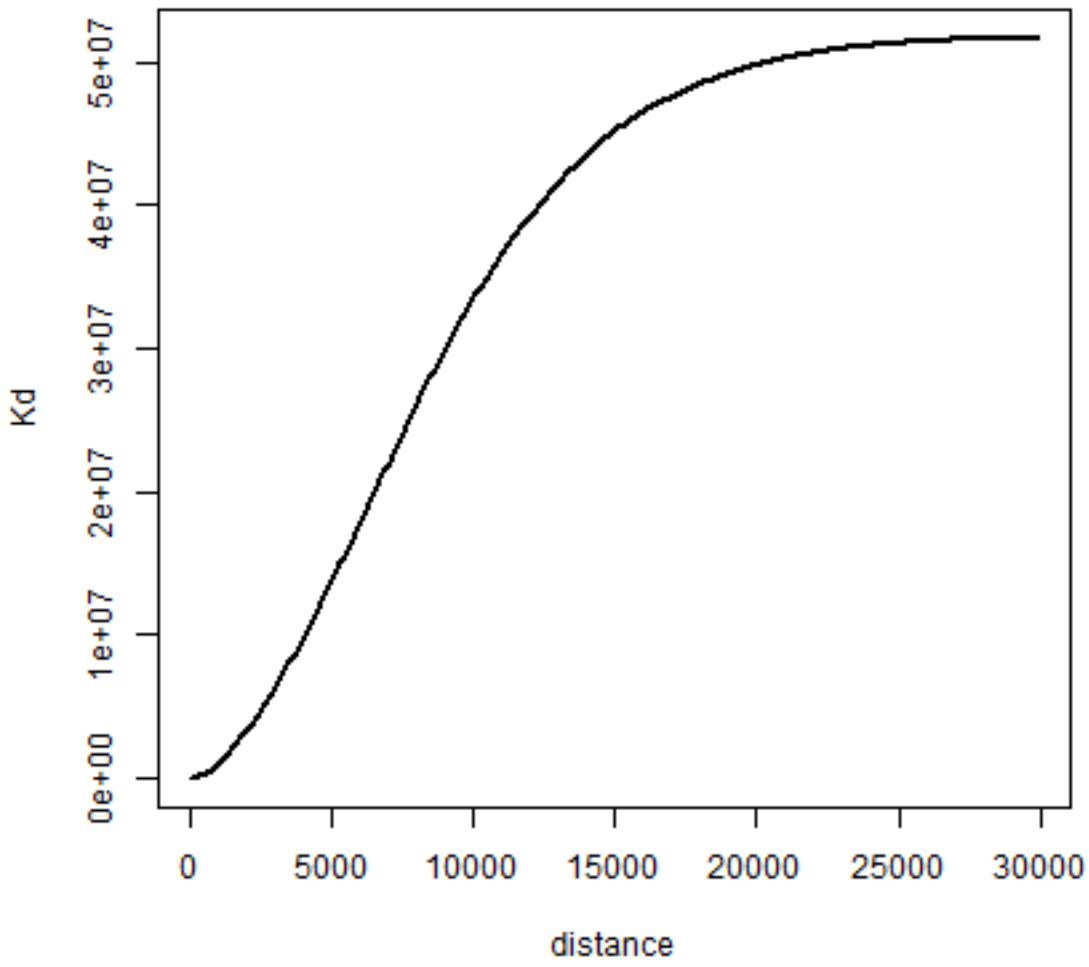
```
legend(1200, .3,
   c(expression(italic("G")["d"]), expression(italic("F")["d"]), 'expected'),
   lty=1, col=c('red', 'blue', 'black'), lwd=2, bty="n")
```



Question 3: *What does this plot suggest about the point pattern?*

Finally, we compute K. Note that I use the original distance matrix d here.

```
distance <- seq(1, 30000, 100)
Kd <- sapply(distance, function(x) sum(d < x)) # takes a while
Kd <- Kd / (length(Kd) * dens)
plot(distance, Kd, type='l', lwd=2)
```



Question 4: *Create a single random pattern of events for the city, with the same number of events as the crime data (object xy). Use function 'terra::spatSample'*

Question 5: *Compute the G function for the observed data, and plot it on a single plot, together with the G function for the theoretical expectation (formula 5.12).*

Question 6: *(Difficult!) Do a Monte Carlo simulation (page 149) to see if the 'mean nearest distance' of the observed crime data is significantly different from a random pattern. Use a 'for loop'. First write 'pseudo-code'. That is, say in natural language what should happen. Then try to write R code that implements this.*

5.5 Spatstat package

Above we did some 'home-brew' point pattern analysis, we will now use the spatstat package. In research you would normally use spatstat rather than your own functions, at least for standard analysis. I showed how you make some of these functions in the previous sections, because understanding how to go about that may allow you to take things in directions that others have not gone. The good thing about spatstat is that it very well documented (see http://spatstat.github.io/). But note that it uses different spatial data classes (ways to represent spatial data) than those that we use elsewhere (classes from sp and raster).

```
library(spatstat)
```

We start with making make a Kernel Density raster. I first create a 'ppp' (point pattern) object, as defined in the `spatstat` package.

A ppp object has the coordinates of the points **and** the analysis 'window' (study region). To assign the points locations we need to extract the coordinates from our SpatVector points object. To set the window, we first need to to coerce our SpatVector polygons into an 'owin' object.

Coerce from SpatVector an object of class "owin" (observation window)

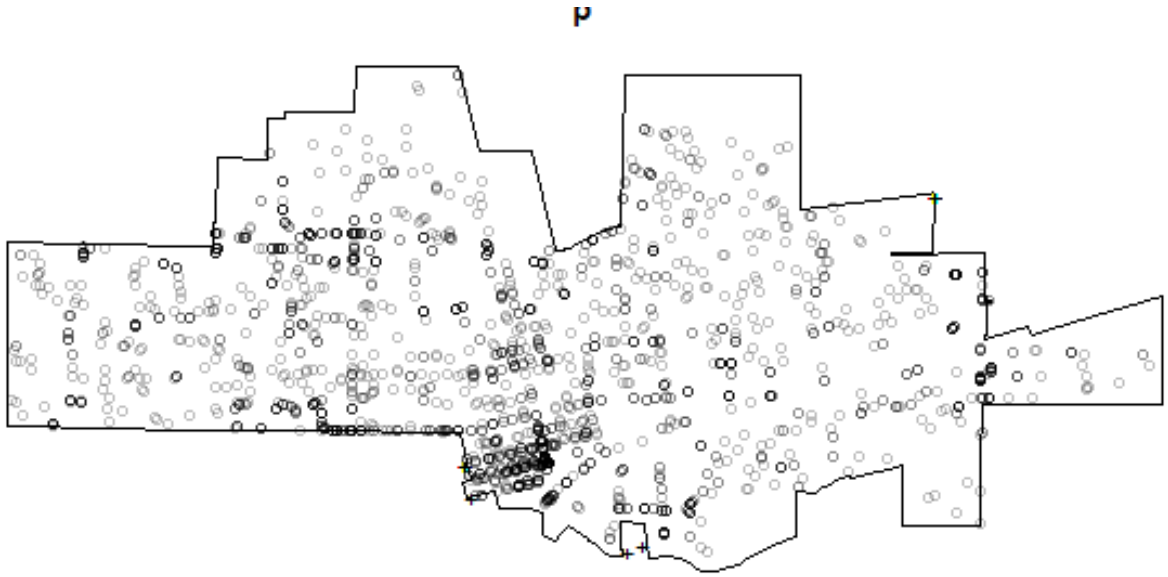
```
cityOwin <- as.owin(sf::st_as_sf(city))
class(cityOwin)
## [1] "owin"
cityOwin
## window: polygonal boundary
## enclosing rectangle: [6620591, 6654380] x [1956729.8, 1971518.9] units
```

Extract coordinates from the SpatVector

```
pts <- geom(crime)[, c("x", "y")]
head(pts)
##            x       y
## [1,] 6628868 1963718
## [2,] 6632796 1964362
## [3,] 6636855 1964873
## [4,] 6626493 1964343
## [5,] 6639506 1966094
## [6,] 6640478 1961983
```

Now we can create a 'ppp' (point pattern) object

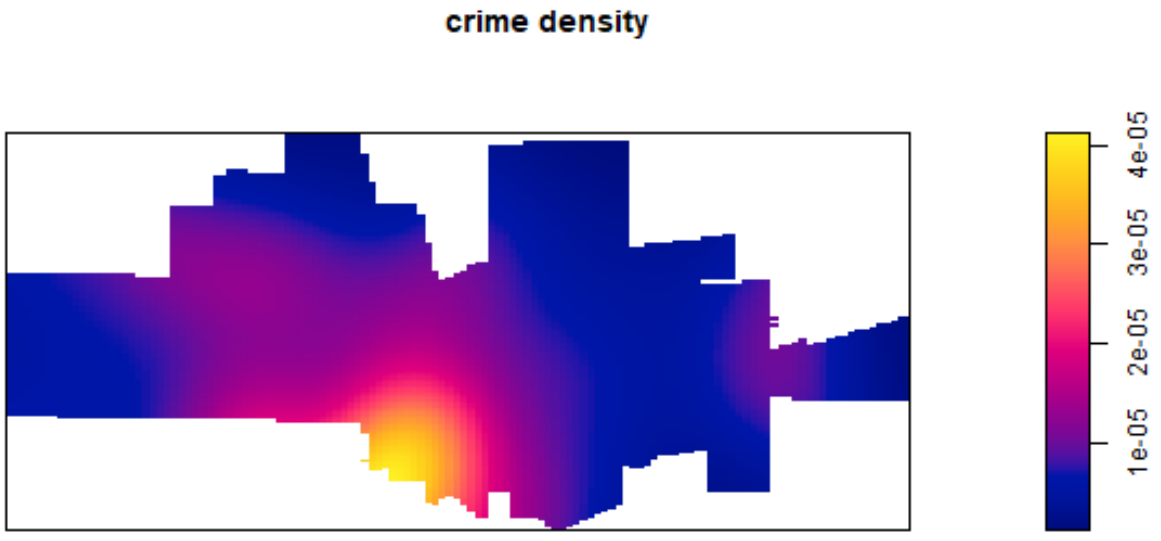
```
p <- ppp(pts[,1], pts[,2], window=cityOwin)
## Warning: 20 points were rejected as lying outside the specified window
## Warning: data contain duplicated points
class(p)
## [1] "ppp"
p
## Planar point pattern: 2641 points
## window: polygonal boundary
## enclosing rectangle: [6620591, 6654380] x [1956729.8, 1971518.9] units
## *** 20 illegal points stored in attr(,"rejects") ***
par(mai=c(0,0,0,0))
plot(p)
## Warning in plot.ppp(p): 20 illegal points also plotted
```



Note the warning message about 'illegal' points. Do you see them and do you understand why they are illegal?

Having all the data well organized, it is now easy to compute Kernel Density

```
ds <- density(p)
class(ds)
## [1] "im"
par(mai=c(0,0,0.5,0.5))
plot(ds, main='crime density')
```



Density is the number of points per unit area. Let's check if the numbers makes sense, by adding them up and multiplying with the area of the raster cells. I use raster package functions for that.

```
nrow(pts)
## [1] 2661
r <- rast(ds)
s <- sum(values(r), na.rm=TRUE)
s * prod(res(r))
## [1] 2640.556
```

Looks about right. We can also get the information directly from the "im" (image) object.

```
str(ds)
## List of 10
##  $ v     : num [1:128, 1:128] NA NA NA NA NA NA NA NA NA NA ...
##  $ dim   : int [1:2] 128 128
##  $ xrange: num [1:2] 6620591 6654380
##  $ yrange: num [1:2] 1956730 1971519
##  $ xstep : num 264
##  $ ystep : num 116
##  $ xcol  : num [1:128] 6620723 6620987 6621251 6621515 6621779 ...
##  $ yrow  : num [1:128] 1956788 1956903 1957019 1957134 1957250 ...
##  $ type  : chr "real"
##  $ units :List of 3
##   ..$ singular  : chr "unit"
##   ..$ plural    : chr "units"
##   ..$ multiplier: num 1
##   ..- attr(*, "class")= chr "unitname"
##  - attr(*, "class")= chr "im"
##  - attr(*, "sigma")= num 1849
##  - attr(*, "kernel")= chr "gaussian"
##  - attr(*, "kerdata")=List of 5
##   ..$ sigma   : num 1849
##   ..$ varcov  : NULL
##   ..$ cutoff  : num 14789
##   ..$ warnings: NULL
##   ..$ kernel  : chr "gaussian"
sum(ds$v, na.rm=TRUE) * ds$xstep * ds$ystep
## [1] 2640.556
p$n
## [1] 2641
```

Here's another, lengthy, example of generalization. We can interpolate population density from (2000) census data; assigning the values to the centroid of a polygon (as explained in the book, but not a great technique). We use a shapefile with census data.

```
census <- spat_data("census2000")
census
##  class       : SpatVector
##  geometry    : polygons
##  dimensions  : 646, 49  (geometries, attributes)
##  extent      : 6576938, 6680926, 1926586, 2007558  (xmin, xmax, ymin, ymax)
##  coord. ref. : +proj=lcc +lat_0=37.6666666666667 +lon_0=-122 +lat_1=38.3333333333333␣
↪+lat_2=39.8333333333333 +x_0=2000000 +y_0=500000.0000000001 +datum=WGS84 +units=us-ft␣
↪+no_defs
##  names       :      CTBLK     CTBNA   CTBLKGP FCCITY   RECNO PLACE STATE COUNTY
```

(continues on next page)

(continued from previous page)

```
##  type        :      <chr>     <num>     <num>  <chr>   <chr> <chr> <chr>  <chr>
##  values      : 0112062000 1.121e+04 1.121e+05 No CDP 0579543 99999    06    113
##                0105052002  1.05e+04 1.051e+05 No CDP 0577237 99999    06    113
##                0112063051 1.121e+04 1.121e+05 No CDP 0579618 99999    06    113
##   TRACT BLOCK (and 39 more)
##    <chr> <chr>
##  011206  2000
##  010505  2002
##  011206  3051
```

To compute population density for each census block, we first need to get the area of each polygon. I transform density from persons per feet^2 to persons per mile^2, and then compute population density from POP2000 and the area

```
census$area <- expanse(census)
census$area <- census$area/27878400
census$dens <- census$POP2000 / census$area
```

Now to get the centroids of the census blocks. Note that it actually does something quite different then in the case above.

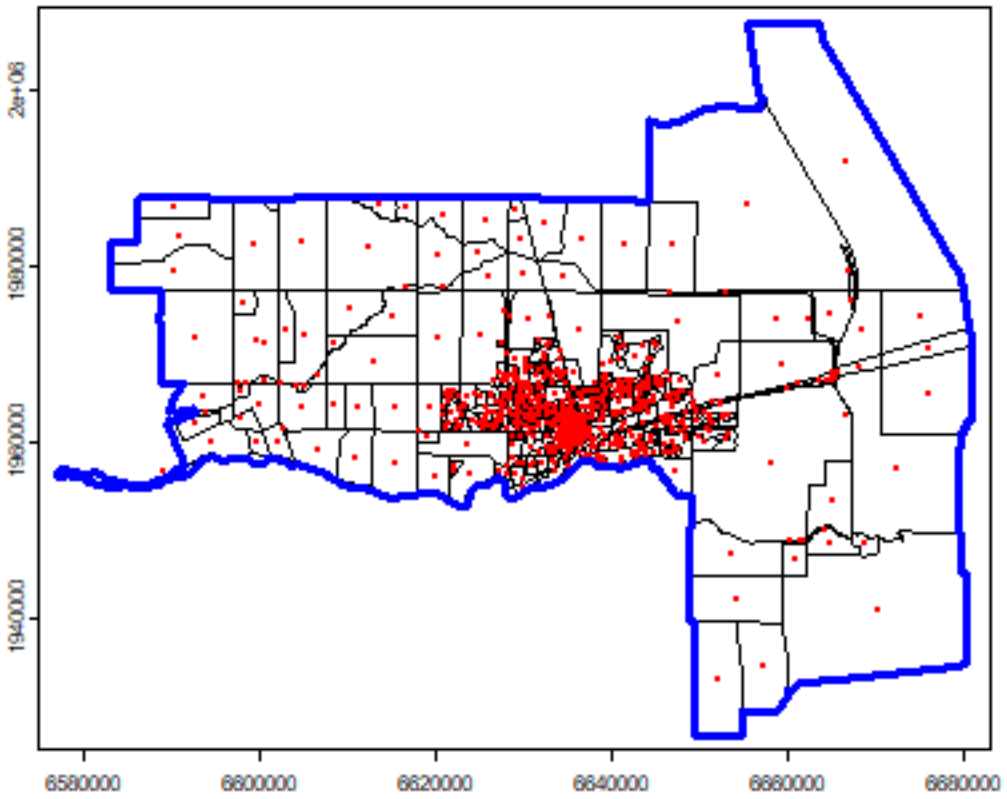
```
p <- centroids(census)
head(p)[, 1:10]
##         CTBLK CTBNA CTBLKGP FCCITY   RECNO PLACE STATE COUNTY  TRACT BLOCK
## 1 0112062000 11206  112062 No CDP 0579543 99999    06    113 011206  2000
## 2 0105052002 10505  105052 No CDP 0577237 99999    06    113 010505  2002
## 3 0112063051 11206  112063 No CDP 0579618 99999    06    113 011206  3051
## 4 0112063049 11206  112063 No CDP 0579616 99999    06    113 011206  3049
## 5 0112063047 11206  112063 No CDP 0579614 99999    06    113 011206  3047
## 6 0112063048 11206  112063 No CDP 0579615 99999    06    113 011206  3048
```

To create the 'window' we dissolve all polygons into a single polygon.

```
win <- aggregate(census)
```

Let's look at what we have:

```
plot(census)
points(p, col='red', pch=20, cex=.25)
plot(win, add=TRUE, border='blue', lwd=3)
```



Now we can use `Smooth.ppp` to interpolate. Population density at the points is referred to as the 'marks'

```
owin <- as.owin(sf::st_as_sf(win))
pxy <- geom(p)[, c("x", "y")]
pp <- ppp(pxy[,1], pxy[,2], window=owin, marks=census$dens)
## Warning: 1 point was rejected as lying outside the specified window
pp
## Marked planar point pattern: 645 points
## marks are numeric, of storage type  'double'
## window: polygonal boundary
## enclosing rectangle: [6576938, 6680926] x [1926586.1, 2007558.2] units
## *** 1 illegal point stored in attr(,"rejects") ***
```

Note the warning message: "1 point was rejected as lying outside the specified window".

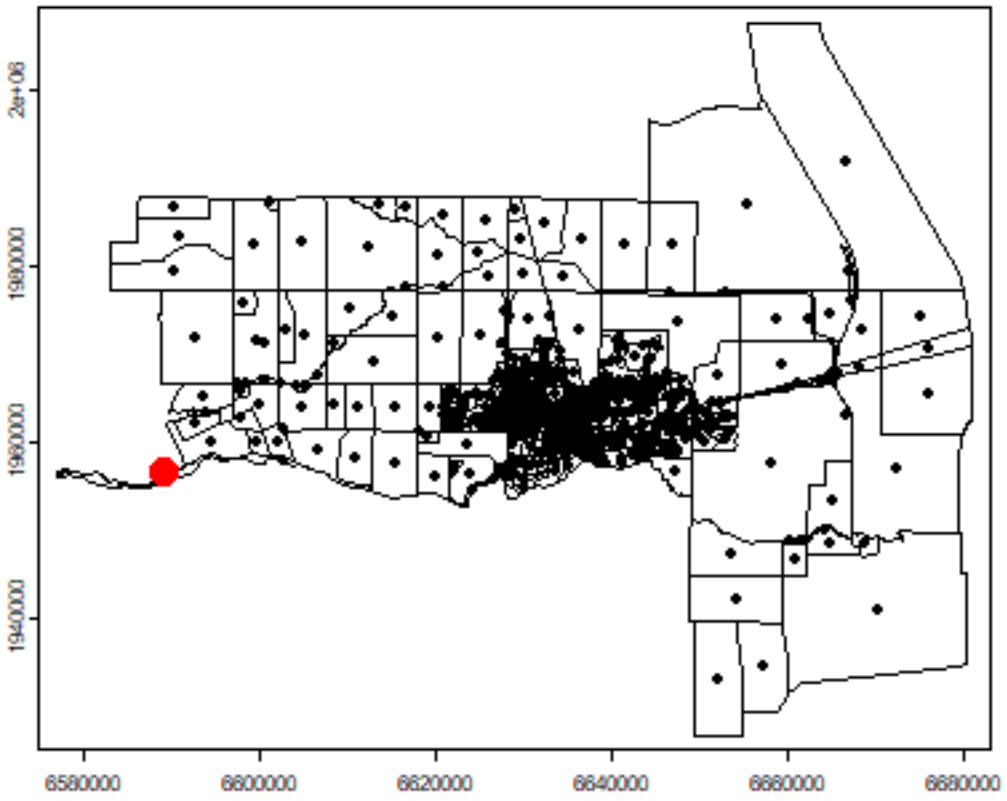
That is odd, there is a polygon that has a centroid that is outside of the polygon. This can happen with, e.g., kidney shaped polygons.

Let's find and remove this point that is outside the study area.

```
i <- relate(p, win, "intersects")
i <- i[,1]
which(!i)
## [1] 588
```

Let's see where it is:

```
plot(census)
points(p)
points(p[!i,], col='red', cex=3, pch=20)
```



You can zoom in using the code below. After running the next line, click on your map twice to zoom to the red dot, otherwise you cannot continue:

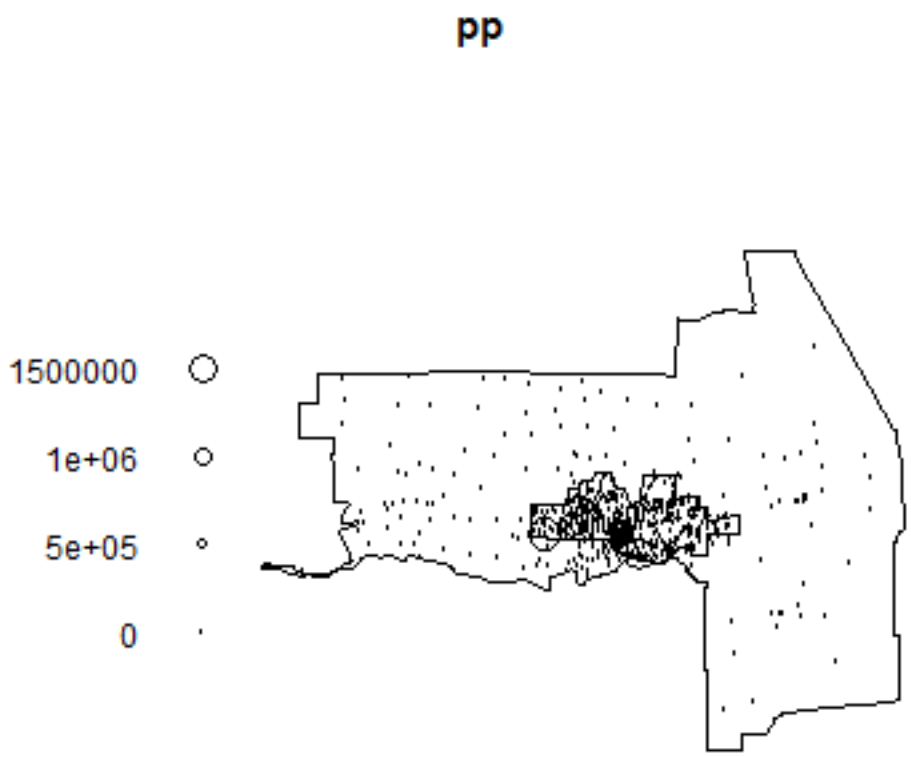
```
zoom(census)
```

And add the red points again

```
points(sp[!i,], col='red')
```

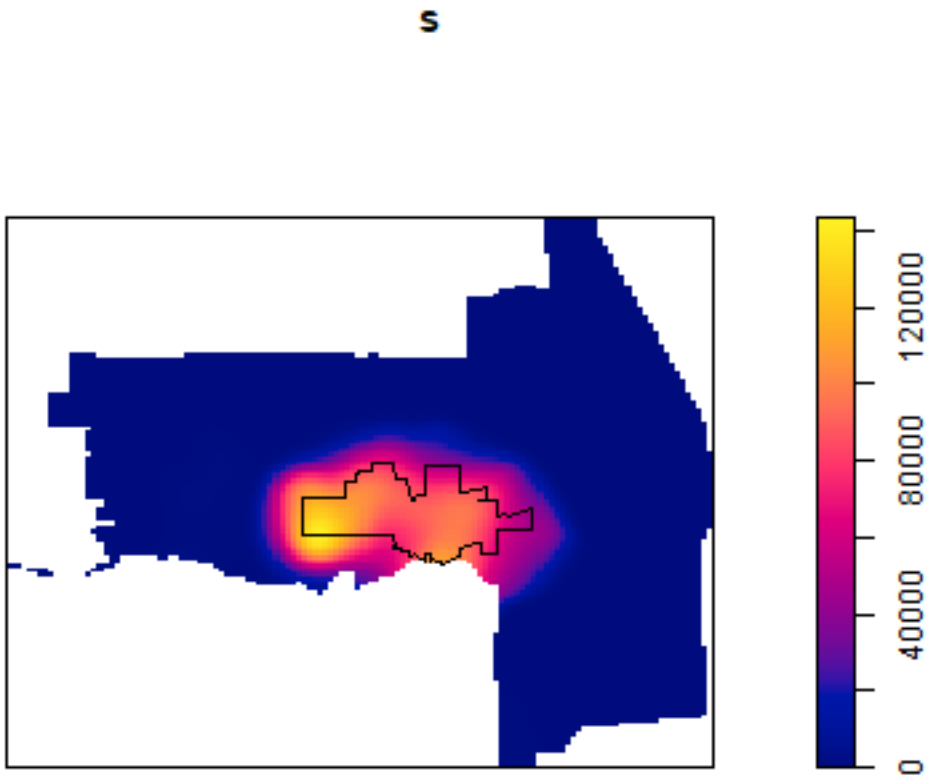
To only use points that intersect with the window polygon, that is, where 'i == TRUE':

```
pp <- ppp(pxy[i,1], pxy[i,2], window=owin, marks=census$dens[i])
plot(pp)
plot(city, add=TRUE)
```



And to get a smooth interpolation of population density.

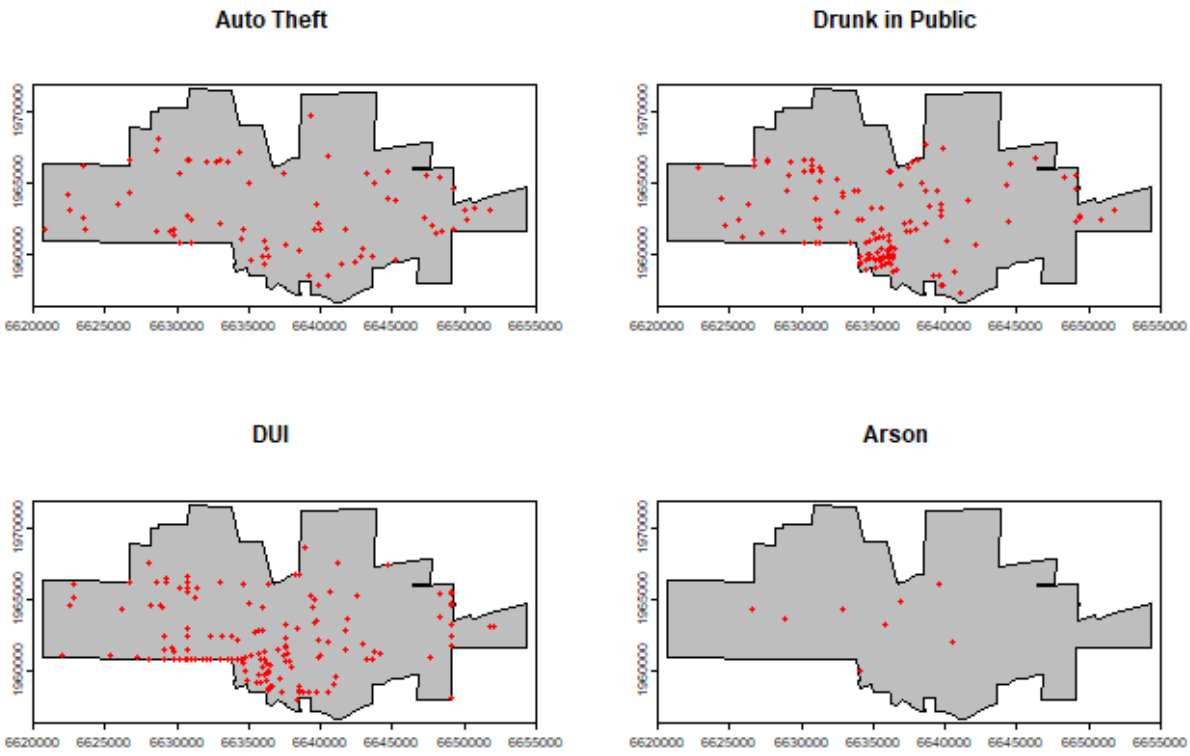
```
s <- Smooth.ppp(pp)
## Warning: Numerical underflow detected: sigma is probably too small
plot(s)
plot(city, add=TRUE)
```



Population density could establish the "population at risk" (to commit a crime) for certain crimes, but not for others.

Maps with the city limits and the incidence of 'auto-theft', 'drunk in public', 'DUI', and 'Arson'.

```
par(mfrow=c(2,2), mai=c(0.25, 0.25, 0.25, 0.25))
for (offense in c("Auto Theft", "Drunk in Public", "DUI", "Arson")) {
  plot(city, col='grey')
    acrime <- crime[crime$CATEGORY == offense, ]
    points(acrime, col = "red")
    title(offense)
}
```

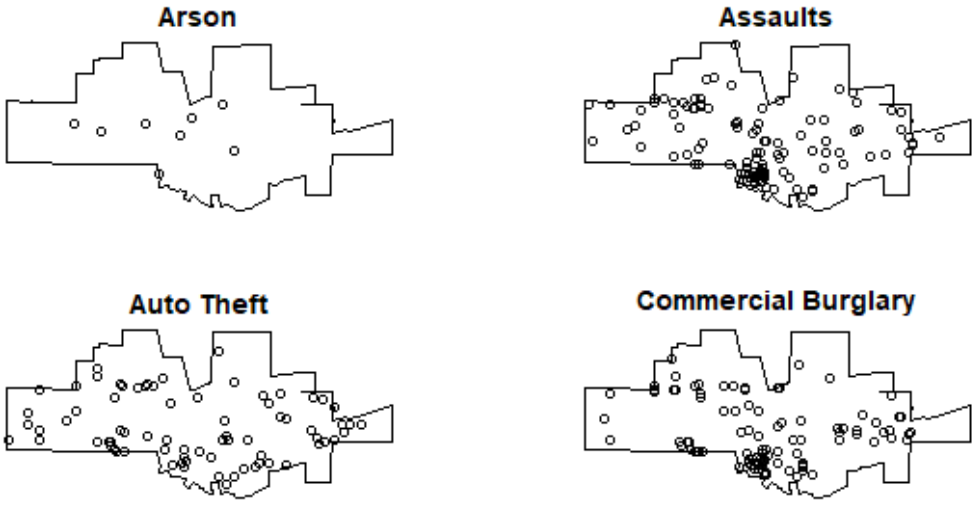


Create a marked point pattern object (ppp) for all crimes. It is important to coerce the marks to a factor variable.

```
fcat <- as.factor(crime$CATEGORY)
cityOwin <- as.owin(sf::st_as_sf(city))
xy <- geom(crime)[, c("x", "y")]
mpp <- ppp(xy[,1], xy[,2], window = cityOwin, marks=fcat)
## Warning: 20 points were rejected as lying outside the specified window
## Warning: data contain duplicated points
```

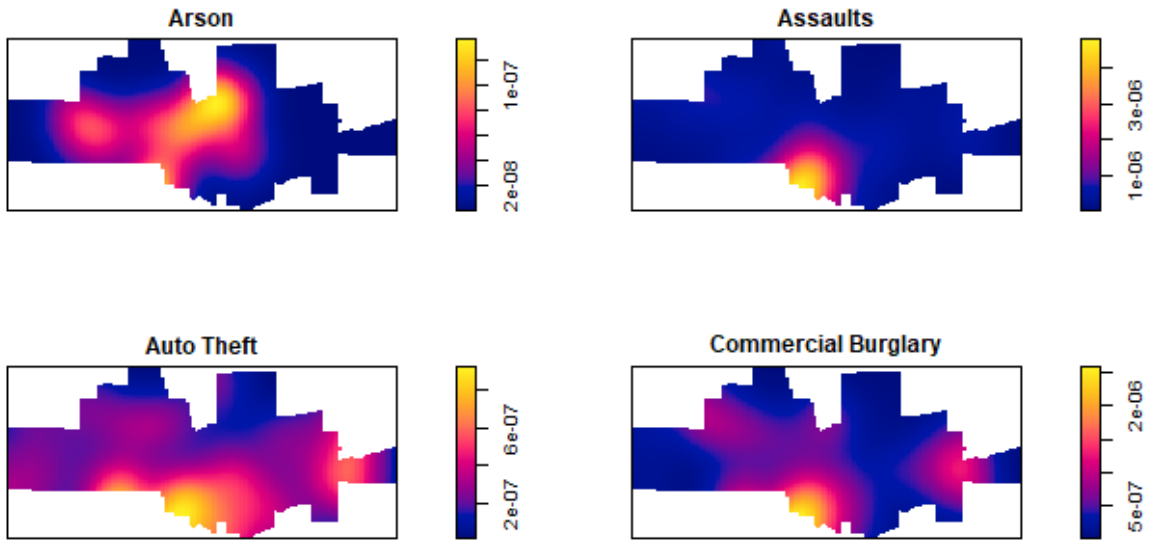
We can split the mpp object by category (crime)

```
par(mai=c(0,0,0,0))
spp <- split(mpp)
plot(spp[1:4], main='')
```



The crime density by category:

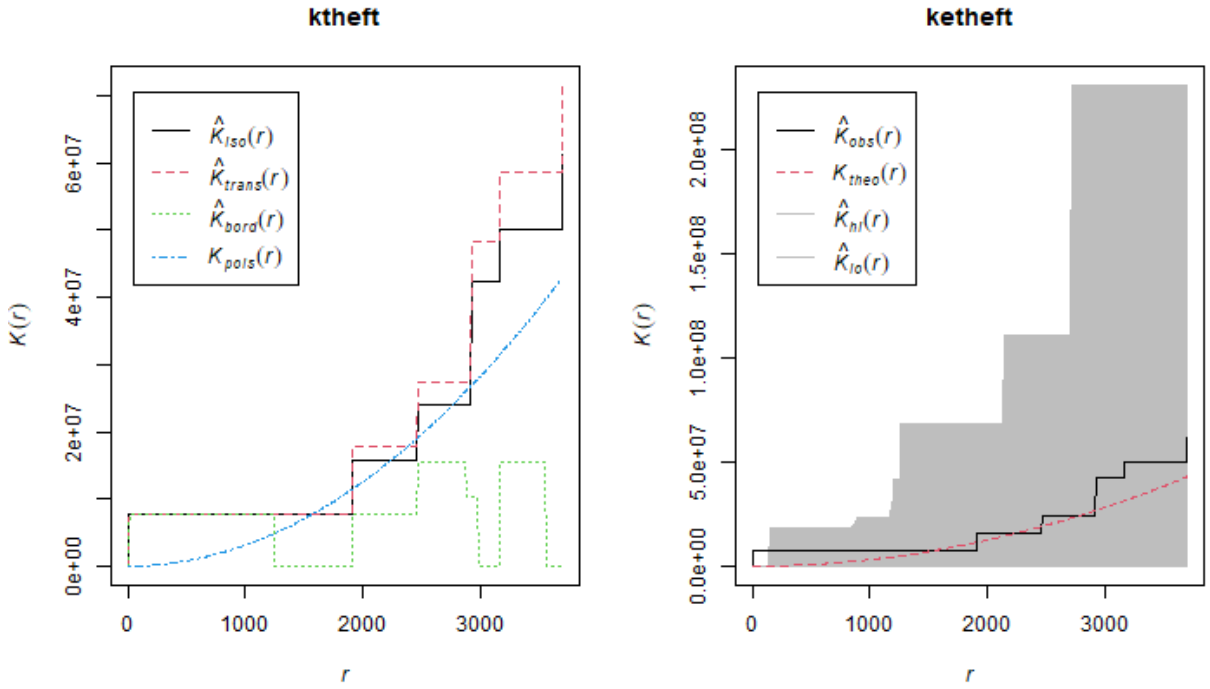
```
plot(density(spp[1:4]), main='')
```



And produce K-plots (with an envelope) for 'drunk in public' and 'Arson'. Can you explain what they mean?

```
spatstat.options(checksegments = FALSE)
ktheft <- Kest(spp$"Auto Theft")
ketheft <- envelope(spp$"Auto Theft", 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.
ktheft <- Kest(spp$"Arson")
ketheft <- envelope(spp$"Arson", 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.
```

```
par(mfrow=c(1,2))
plot(ktheft)
plot(ketheft)
```



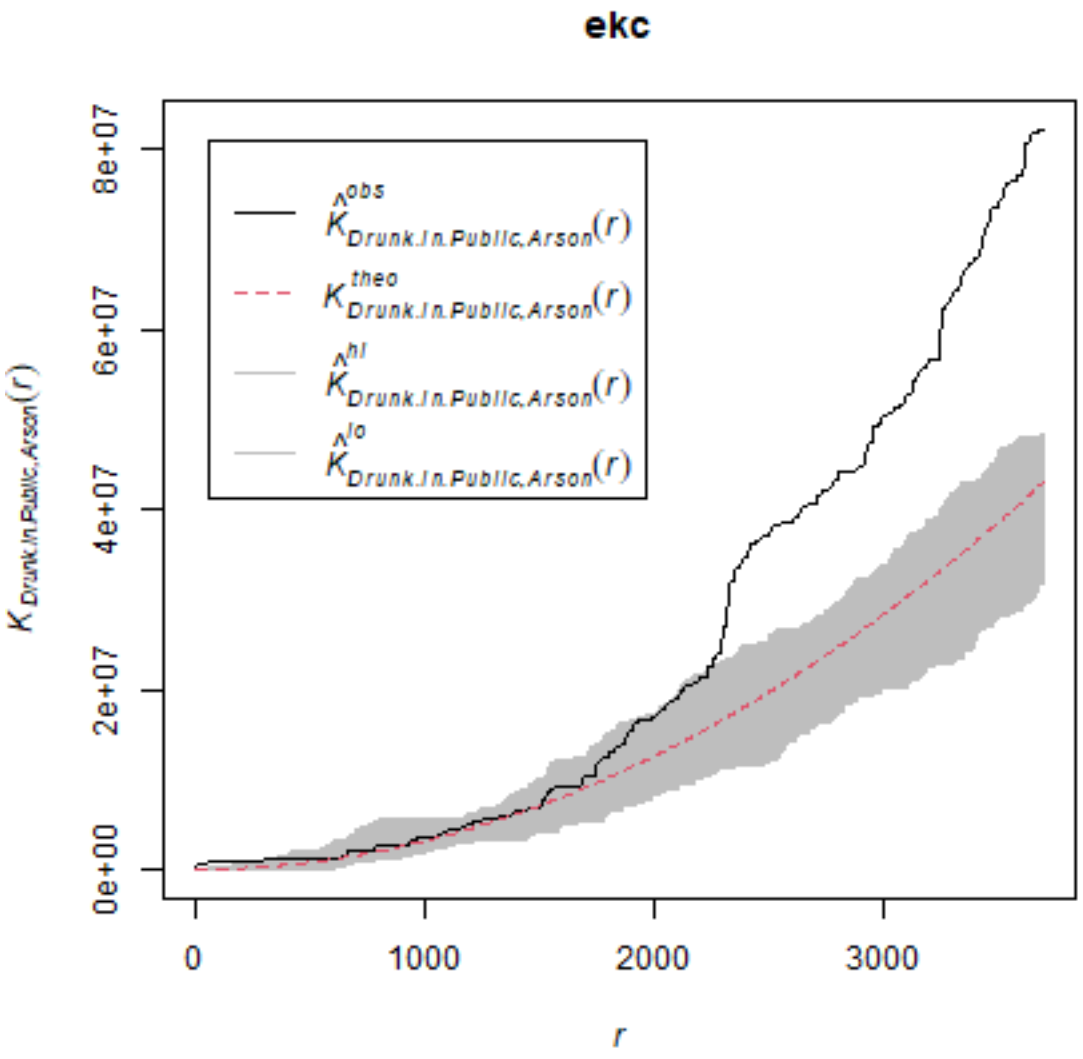
Let's try to answer the question you have been wanting to answer all along. Is population density a good predictor of being (booked for) "drunk in public" and for "Arson"? One approach is to do a Kolmogorov-Smirnov test on 'Drunk in Public' and 'Arson', using population density as a covariate:

```
KS.arson <- cdf.test(spp$Arson, covariate=ds, test='ks')
KS.arson
##
##  Spatial Kolmogorov-Smirnov test of CSR in two dimensions
##
## data:  covariate 'ds' evaluated at points of 'spp$Arson'
##      and transformed to uniform distribution under CSR
## D = 0.50391, p-value = 0.0122
## alternative hypothesis: two-sided
KS.drunk <- cdf.test(spp$'Drunk in Public', covariate=ds, test='ks')
KS.drunk
##
##  Spatial Kolmogorov-Smirnov test of CSR in two dimensions
##
## data:  covariate 'ds' evaluated at points of 'spp$"Drunk in Public"'
##      and transformed to uniform distribution under CSR
## D = 0.54021, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

Question 7: *Why is the result surprising, or not surprising?*

We can also compare the patterns for "drunk in public" and for "Arson" with the KCross function.

```
kc <- Kcross(mpp, i = "Drunk in Public", j = "Arson")
ekc <- envelope(mpp, Kcross, nsim = 50, i = "Drunk in Public", j = "Arson")
## Generating 50 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.
##
## Done.
plot(ekc)
```




```
detach("package:spatstat", unload=TRUE)
```

CHAPTER

SIX

SPATIAL AUTOCORRELATION

6.1 Introduction

This handout accompanies Chapter 7 in O'Sullivan and Unwin (2010).

6.2 The area of a polygon

Create a polygon like in Figure 7.2 (page 192).

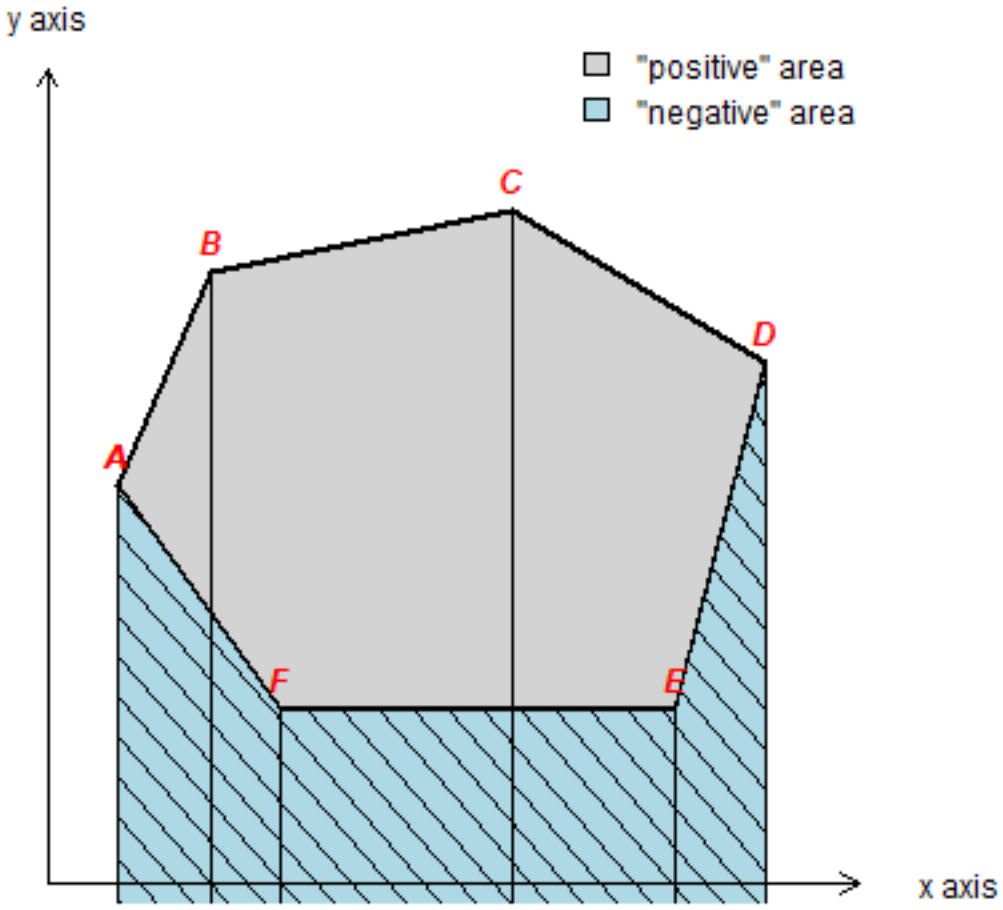
```
library(terra)
## terra 1.7.62
pol <- matrix(c(1.7, 2.6, 5.6, 8.1, 7.2, 3.3, 1.7, 4.9, 7, 7.6, 6.1, 2.7, 2.7, 4.9),␣
↪ncol=2)
sppol <- vect(pol, "polygons")
```

For illustration purposes, we create the "negative area" polygon as well

```
negpol <- rbind(pol[c(1,6:4),], cbind(pol[4,1], 0), cbind(pol[1,1], 0))
spneg <- vect(negpol, "polygons")
```

Now plot

```
cols <- c('light gray', 'light blue')
plot(sppol, xlim=c(1,9), ylim=c(1,10), col=cols[1], axes=FALSE, xlab='', ylab='',
      lwd=2, yaxs="i", xaxs="i")
plot(spneg, col=cols[2], add=T)
plot(spneg, add=T, density=8, angle=-45, lwd=1)
segments(pol[,1], pol[,2], pol[,1], 0)
text(pol, LETTERS[1:6], pos=3, col='red', font=4)
arrows(1, 1, 9, 1, 0.1, xpd=T)
arrows(1, 1, 1, 9, 0.1, xpd=T)
text(1, 9.5, 'y axis', xpd=T)
text(10, 1, 'x axis', xpd=T)
legend(6, 9.5, c('"positive" area', '"negative" area'), fill=cols, bty = "n")
```



Compute area

```
p <- rbind(pol, pol[1,])
x <- p[-1,1] - p[-nrow(p),1]
y <- (p[-1,2] + p[-nrow(p),2]) / 2
sum(x * y)
## [1] 23.81
```

Or simply use an existing function. To make sure that the coordinates are not interpreted as longitude/latitude I assign an arbitrary planar coordinate reference system.

```
crs(sppol) <- '+proj=utm +zone=1'
expanse(sppol)
## [1] 23.68211
```

6.3 Contact numbers

"Contact numbers" for the lower 48 states. Get the polygons using the `geodata` package:

```
if (!require(geodata)) remotes::install_github('rspatial/geodata')
## Loading required package: geodata
usa <- geodata::gadm(country='USA', level=1, path=".")
usa <- usa[! usa$NAME_1 %in% c('Alaska', 'Hawaii'), ]
```

To find adjacent polygons, we can use the `relate` method.

```
# patience, this takes a while:
wus <- relate(usa, relation="touches")
rownames(wus) <- colnames(wus) <- usa$NAME_1
wus[1:5,1:5]
##            Alabama Arizona Arkansas California Colorado
## Alabama      FALSE   FALSE    FALSE      FALSE    FALSE
## Arizona      FALSE   FALSE    FALSE       TRUE     TRUE
## Arkansas     FALSE   FALSE    FALSE      FALSE    FALSE
## California   FALSE    TRUE    FALSE      FALSE    FALSE
## Colorado     FALSE    TRUE    FALSE      FALSE    FALSE
```

Compute the number of neighbors for each state.

```
i <- rowSums(wus)
round(100 * table(i) / length(i), 1)
## i
##    1    2    3    4    5    6    7    8
##  2.0 10.2 16.3 18.4 18.4 26.5  4.1  4.1
```

Apparently, I am using a different data set than OSU (compare the above with table 7.1). By changing the `level` argument to 2 in the `getData` function you can run the same for counties.

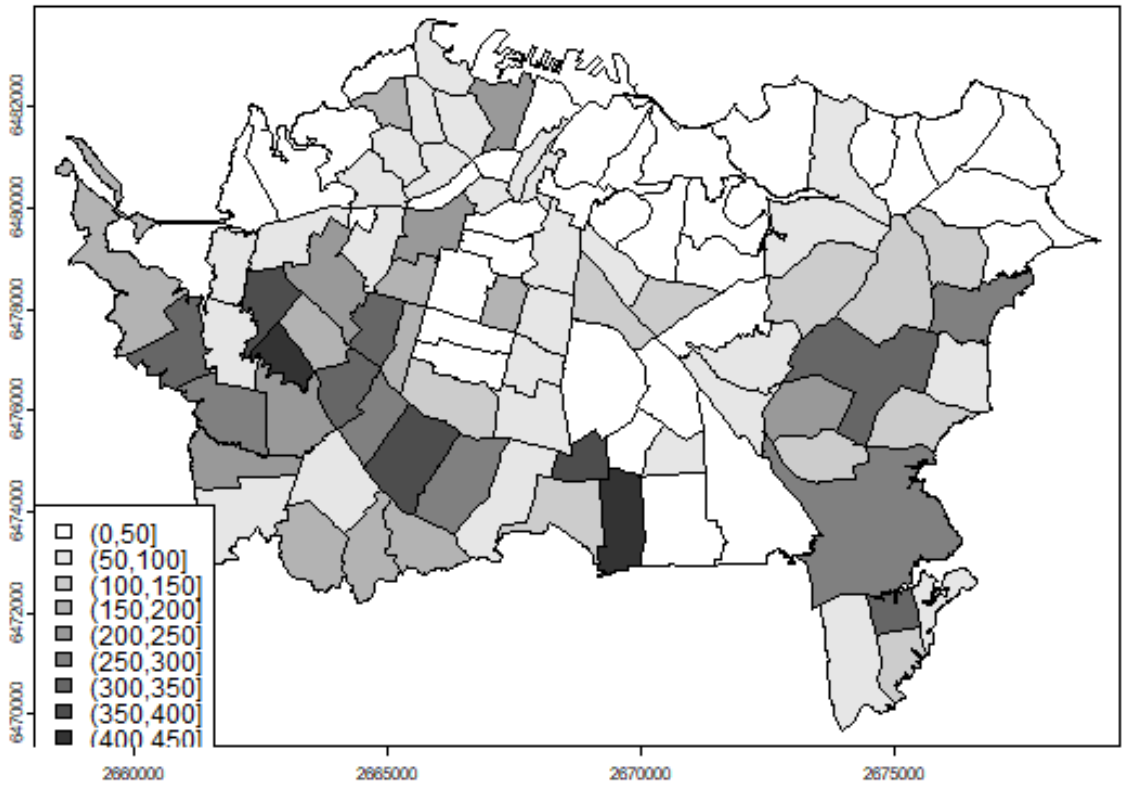
6.4 Spatial structure

Read the Auckland data from the `rspat` package

```
if (!require("rspat")) remotes::install_github('rspatial/rspat')
## Loading required package: rspat
library(rspat)
pols <- spat_data("auctb")
```

The tuberculosis data used here were estimated them from figure 7.7. Compare:

```
par(mai=c(0,0,0,0))
classes <- seq(0,450,50)
cuts <- cut(pols$TB, classes)
n <- length(classes)
cols <- rev(gray(0:n / n))
plot(pols, col=cols[as.integer(cuts)])
legend('bottomleft', levels(cuts), fill=cols)
```

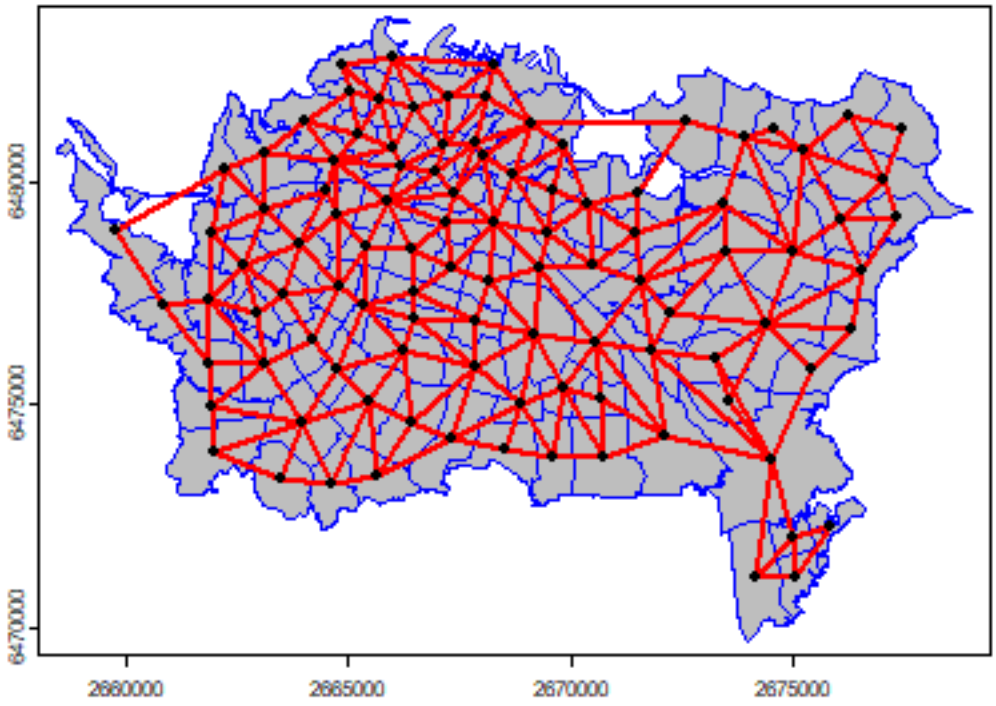


Find "rook" connetected areas.

```
wr <- adjacent(pols, type="rook", symmetrical=TRUE)
head(wr)
##      from to
## [1,]    1 28
## [2,]    1 53
## [3,]    1 73
## [4,]    1 74
## [5,]    1 75
## [6,]    1 76
```

Plot the links between the polygons.

```
v <- centroids(pols)
p1 <- v[wr[,1], ]
p2 <- v[wr[,2], ]

par(mai=c(0,0,0,0))
plot(pols, col='gray', border='blue')
lines(p1, p2, col='red', lwd=2)
points(v)
```



Now let's recreate Figure 7.6 (page 202).

We already have the first one (Rook's case adjacency, plotted above). Queen's case adjacency:

```
wq <- adjacent(pols, "queen", symmetrical=TRUE)
```

Distance based:

```
wd1 <- nearby(pols, distance=1000)
wd25 <- nearby(pols, distance=2500)
```

Nearest neighbors:

```
k3 <- nearby(pols, k=3)
k6 <- nearby(pols, k=6)
```

Delauny:

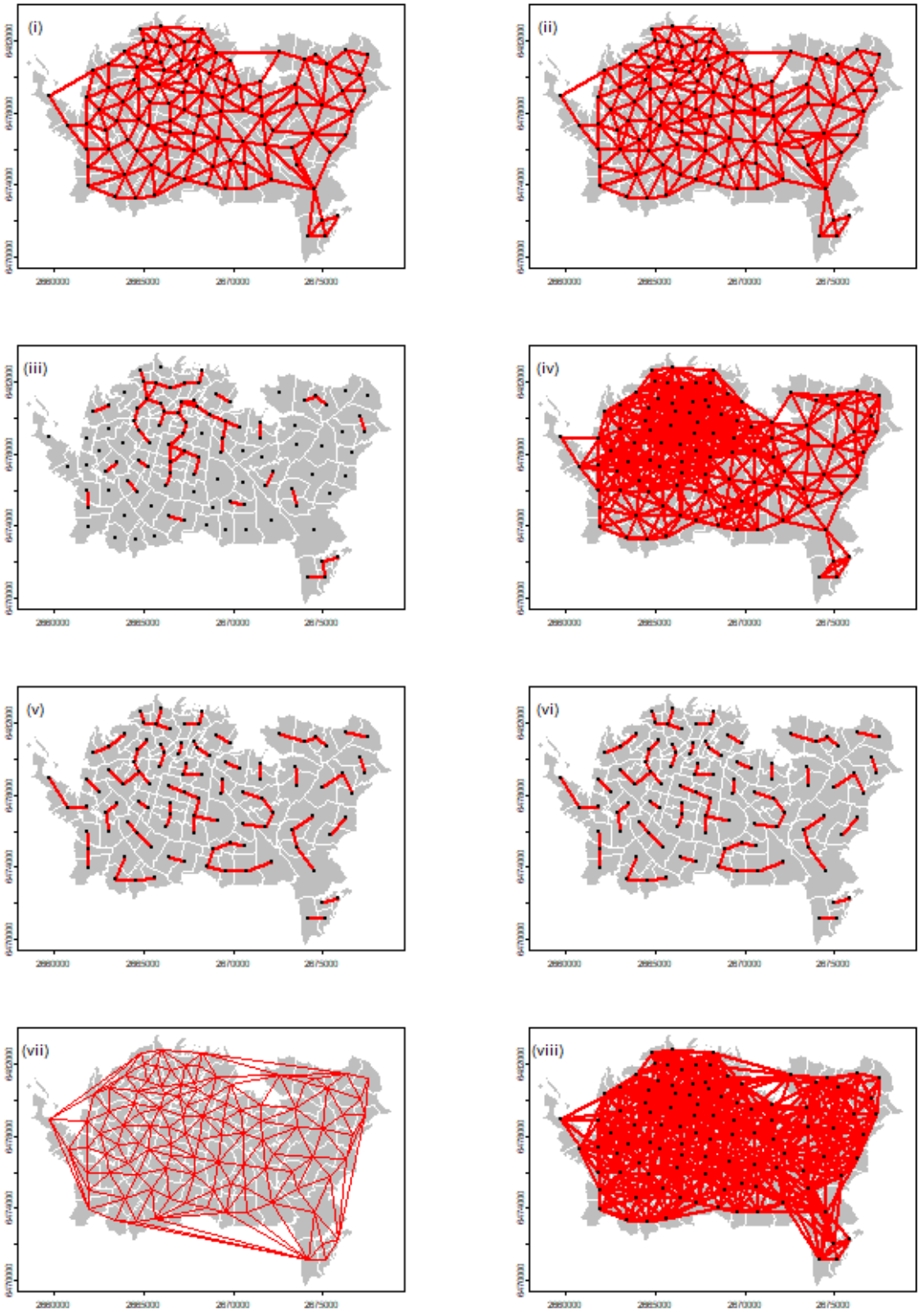
```
d <- delaunay(centroids(pols))
```

Lag-two Rook:

```
wrs <- adjacent(pols, "rook", symmetrical=FALSE)
uf <- sort(unique(wrs[,1]))
wr2 <- list()
for (i in 1:length(pols)) {
    lag1 <- wrs[wrs[,1]==i, 2]
  lag2 <- wrs[wrs[,1] %in% lag1, ]
  lag2[,1] <- i
    wr2[[i]] <- unique(lag2)
}

wr2 <- do.call(rbind, wr2)
```

And now we plot them all using the `plotit` function.

```
plotit <- function(nb, lab='') {
  plot(pols, col='gray', border='white')
  v <- centroids(pols)
  p1 <- v[nb[,1], ]
  p2 <- v[nb[,2], ]
  lines(p1, p2, col='red', lwd=2)
  points(v)
  text(2659066, 6482808, paste0('(', lab, ')'), cex=1.25)
}

par(mfrow=c(4, 2), mai=c(0,0,0,0))
plotit(wr, 'i')
plotit(wq, 'ii')
plotit(wd1, 'iii')
plotit(wd25, 'iv')
plotit(k3, 'v')
plotit(k6, 'vi')
plot(pols, col='gray', border='white')
lines(d, col="red")
text(2659066, 6482808, '(vii)', cex=1.25)
plotit(wr2, 'viii')
```


(i)
(ii)
(iii)
(iv)
(v)
(vi)
(vii)
(viii)

6.5 Moran's *I*

Below I compute Moran's index according to formula 7.7 on page 205 of OSU.

$$I = \frac{n}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(y_i - \bar{y})(y_j - \bar{y})}{\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}}$$

The number of observations

```
n <- length(pols)
```

Values 'y' and 'ybar' (the mean of y).

```
y <- pols$TB
ybar <- mean(y)
```

Now we need

$$(y_i - \bar{y})(y_j - \bar{y})$$

That is, (yi-ybar)(yj-ybar) for all pairs. I show two methods to compute that.

Method 1:

```
dy <- y - ybar
g <- expand.grid(dy, dy)
yiyj <- g[,1] * g[,2]
```

Method 2:

```
yi <- rep(dy, each=n)
yj <- rep(dy)
yiyj <- yi * yj
```

Make a matrix of the multiplied pairs

```
pm <- matrix(yiyj, ncol=n)
round(pm[1:6, 1:9])
##        [,1]  [,2]   [,3]   [,4]  [,5]   [,6]   [,7]   [,8]  [,9]
## [1,]  22778  4214 -12538  30776 -3181 -10727 -17821 -12387 -5445
## [2,]   4214   780  -2320   5694  -589  -1985  -3297  -2292 -1007
## [3,] -12538 -2320   6902 -16941  1751   5905   9810   6819  2997
## [4,]  30776  5694 -16941  41584 -4298 -14494 -24079 -16737 -7357
## [5,]  -3181  -589   1751  -4298   444   1498   2489   1730   760
## [6,] -10727 -1985   5905 -14494  1498   5052   8393   5834  2564
```

And multiply this matrix with the weights to set to zero the value for the pairs that are not adjacent.

```
wm <- adjacent(pols, "rook", pairs=FALSE)
wm[1:9, 1:11]
##   1 2 3 4 5 6 7 8 9 10 11
## 1 0 0 0 0 0 0 0 0 0  0  0
## 2 0 0 0 0 0 0 0 0 0  0  0
## 3 0 0 0 0 0 0 0 0 0  0  0
## 4 0 0 0 0 0 0 0 0 0  0  0
```

(continues on next page)

(continued from previous page)

```
## 5 0 0 0 0 0 0 0 0 0  0  0
## 6 0 0 0 0 0 0 1 0 0  0  1
## 7 0 0 0 0 0 1 0 0 0  0  1
## 8 0 0 0 0 0 0 0 0 1  0  0
## 9 0 0 0 0 0 0 0 1 0  0  0
pmw <- pm * wm
round(pmw[1:9, 1:11])
##   1 2 3 4 5    6    7    8    9 10    11
## 1 0 0 0 0 0    0    0    0    0  0     0
## 2 0 0 0 0 0    0    0    0    0  0     0
## 3 0 0 0 0 0    0    0    0    0  0     0
## 4 0 0 0 0 0    0    0    0    0  0     0
## 5 0 0 0 0 0    0    0    0    0  0     0
## 6 0 0 0 0 0    0 8393    0    0  0  6616
## 7 0 0 0 0 0 8393    0    0    0  0 10990
## 8 0 0 0 0 0    0    0    0 2961  0     0
## 9 0 0 0 0 0    0    0 2961    0  0     0
```

We sum the values, to get this bit of Moran's I:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(y_i - \bar{y})(y_j - \bar{y})$$

```
spmw <- sum(pmw)
spmw
## [1] 1523422
```

The next step is to divide this value by the sum of weights. That is easy.

```
smw <- sum(wm)
sw  <- spmw / smw
```

And compute the inverse variance of y

```
vr <- n / sum(dy^2)
```

The final step to compute Moran's I

```
MI <- vr * sw
MI
## [1] 0.2643226
```

After doing this 'by hand', now let's use the terra package to compute Moran's I.

```
autocor(y, wm)
## [1] 0.2643226
```

This is how you can (theoretically) estimate the expected value of Moran's I. That is, the value you would get in the absence of spatial autocorrelation. Note that it is not zero for small values of n.

```
  EI <- -1/(n-1)
EI
## [1] -0.009803922
```

So is the value we found significant, in the sense that is it not a value you would expect to find by random chance? Significance can be tested analytically (see `spdep::moran.test`) but it is much better to use Monte Carlo simulation. We test the (one-sided) probability of getting a value as high as the observed I.

```
I <- autocor(pols$TB, wm)
nsim <- 99
mc <- sapply(1:nsim, function(i) autocor(sample(pols$TB), wm))

P <- 1 - sum((I > mc) / (nsim+1))
P
## [1] 0.01
```

Question 1: *How do you interpret these results (the significance tests)?*

Question 2: *What would a good value be for ``nsim``?*

Question 3: *Show a figure similar to Figure 7.9 in OSU.

To make a Moran scatter plot we first get the neighbouring values for each value.

```
n <- length(pols)
ms <- cbind(id=rep(1:n, each=n), y=rep(y, each=n), value=as.vector(wm * y))
```

Remove the zeros

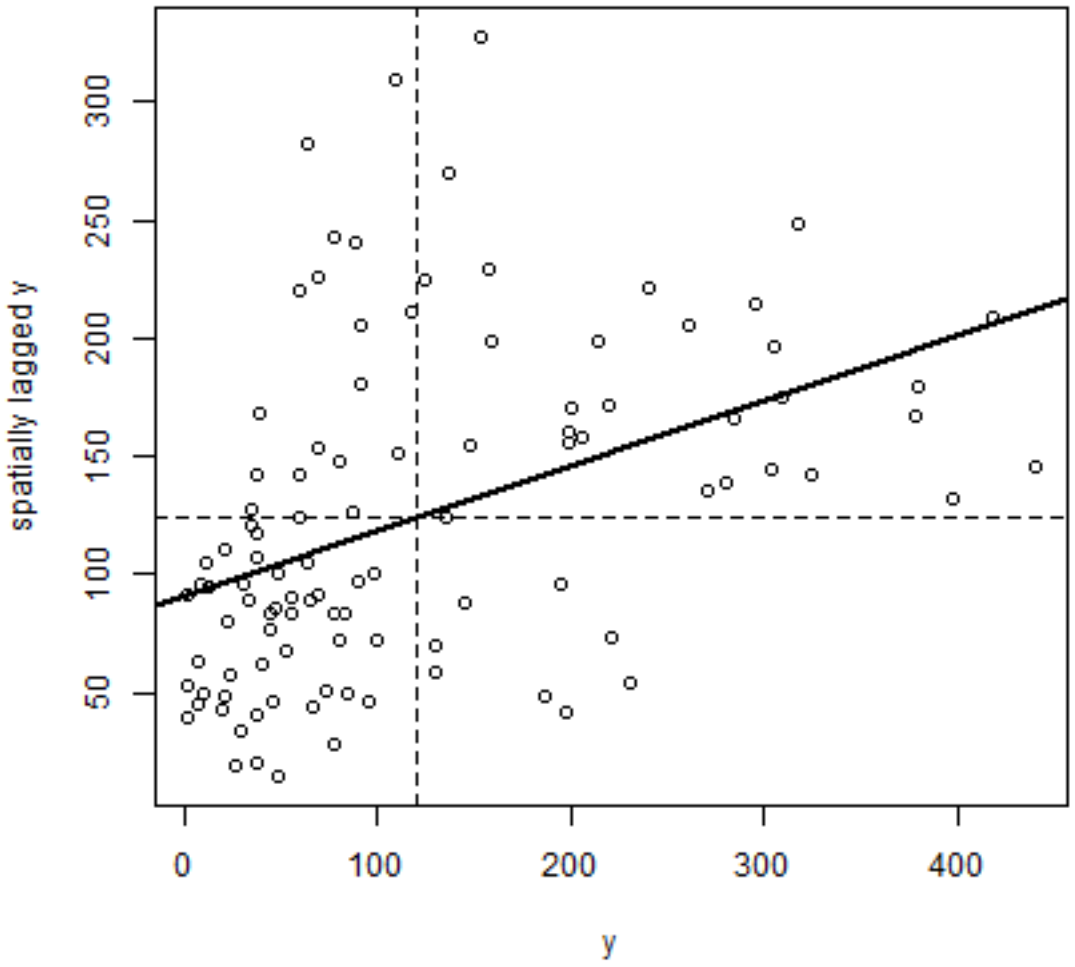
```
ms <- ms[ms[,3] > 0, ]
```

And compute the average neighbour value

```
ams <- aggregate(ms[,2:3], list(ms[,1]), FUN=mean)
ams <- ams[,-1]
colnames(ams) <- c('y', 'spatially lagged y')
head(ams)
##     y spatially lagged y
## 1 271           135.1429
## 2 148           154.3333
## 3  37           142.2500
## 4 324           142.6250
## 5  99            71.6250
## 6  49            14.5000
```

Finally, the plot.

```
plot(ams)
reg <- lm(ams[,2] ~ ams[,1])
abline(reg, lwd=2)
abline(h=mean(ams[,2]), lt=2)
abline(v=ybar, lt=2)
```



Note that the slope of the regression line:

```
coefficients(reg)[2]
##  ams[, 1]
## 0.2746281
```

is almost the same as Moran's *I*.

See `spdep::moran.plot` for a more direct approach to accomplish the same thing (but hopefully the above makes it clearer how this is actually computed).

Question 4: *compute Geary's C for these data*

CHAPTER

SEVEN

LOCAL STATISTICS

7.1 Introduction

This handout accompanies Chapter 8 in O'Sullivan and Unwin (2010).

7.2 LISA

We compute some measures of local spatial autocorrelation.

First get the Auckland data.

```
if (!require("rspat")) remotes::install_github("rspatial/rspat")
library(rspat)
auck <- spat_data("auctb")
```

Now compute the spatial weights. You can try other ways (e.g. `relation="touches"`)

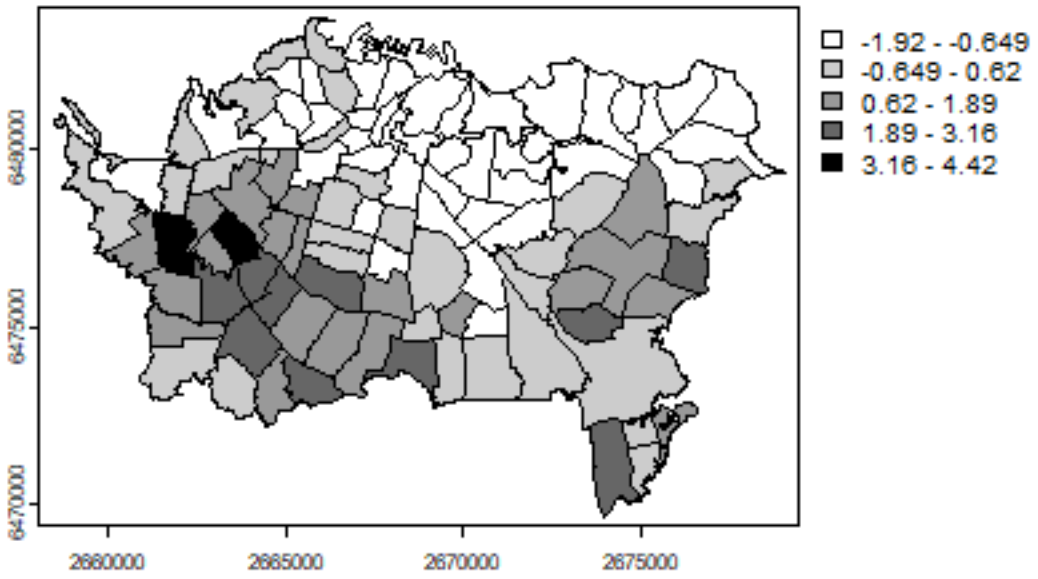
```
w <- relate(auck, relation="rook")
```

Compute the Getis *Gi*

```
Gi <- autocor(auck$TB, w, "Gi")
head(Gi)
## [1]  0.4241759  0.5623307  0.4047413  0.6819762 -1.3278352 -1.4086435
```

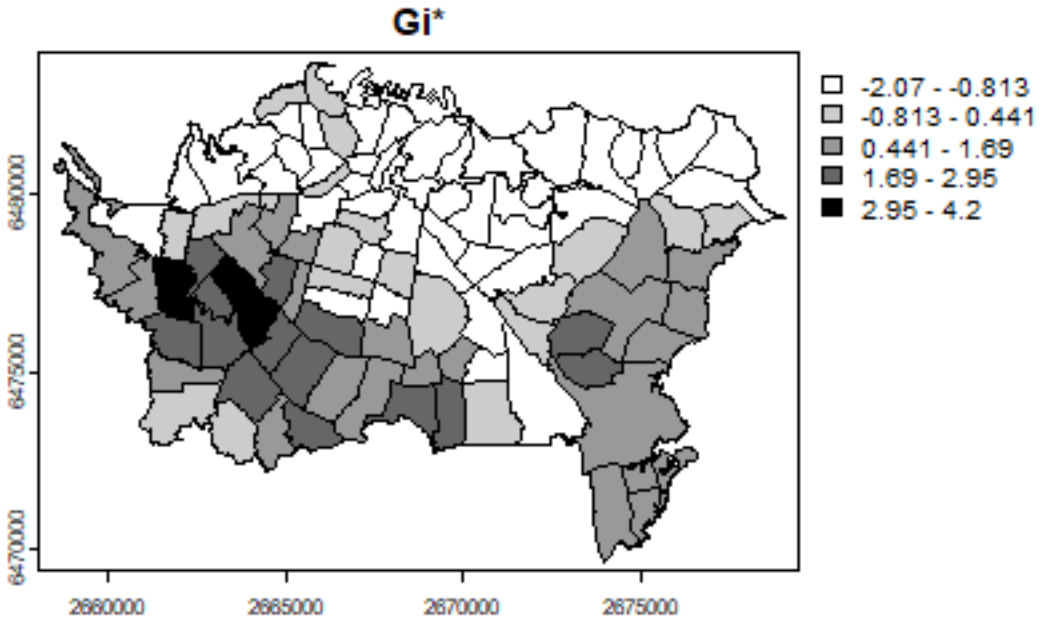
And make a map

```
grays <- rev(gray(seq(0,1,.2)))
auck$Gi <- Gi
plot(auck, "Gi", col=grays)
```



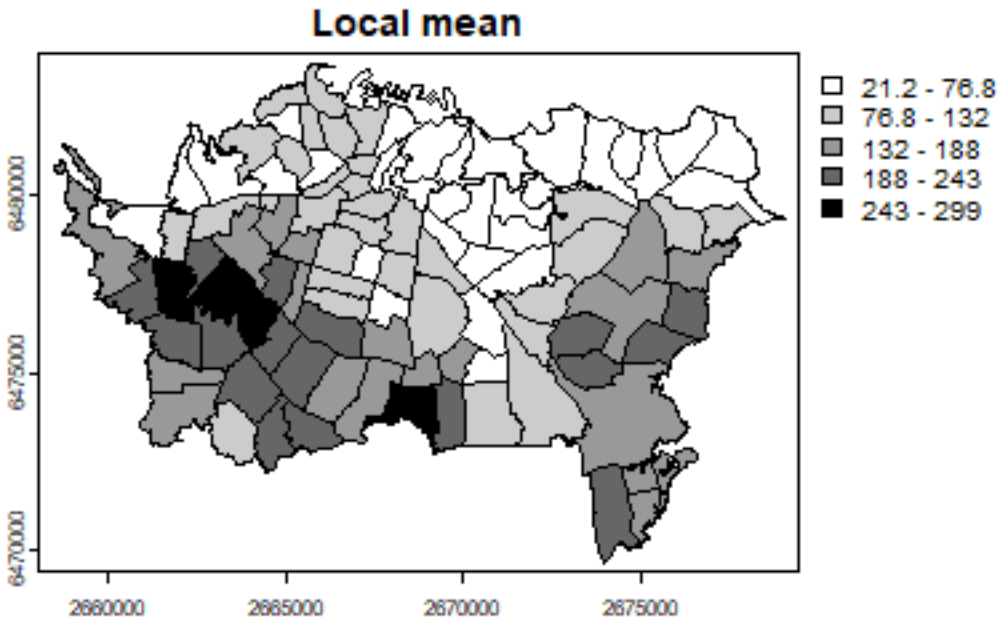
Now on to the `Gi*` by including the focal area in the computation.

```
#include "self"
diag(w) <- TRUE
auck$Gistar <- autocor(auck$TB, w, "Gi*")
plot(auck, "Gistar", main="Gi*", col=grays)
```



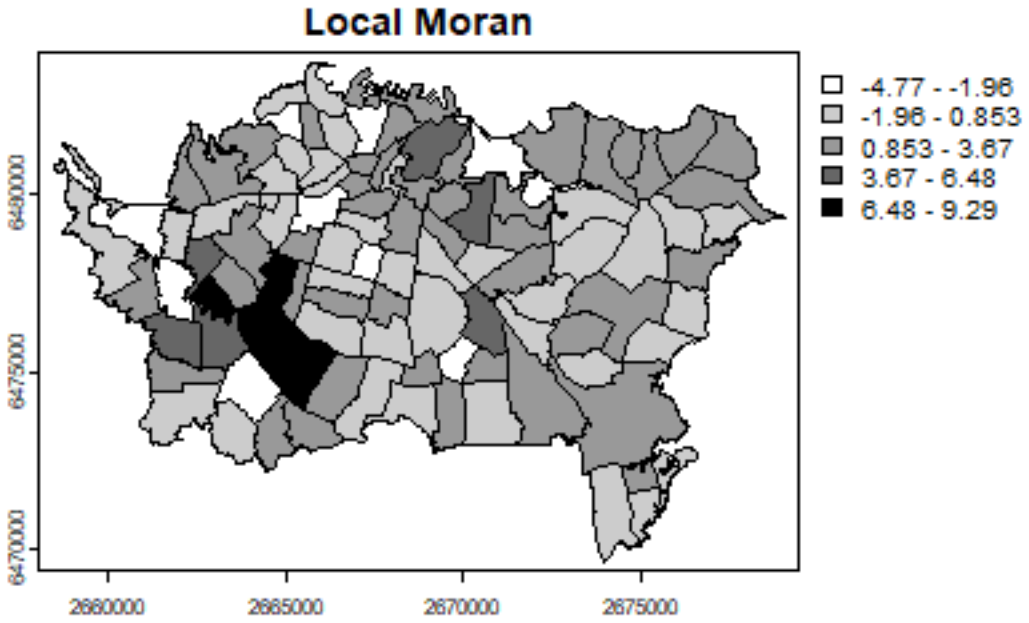
This looks very similar to the local average, which we compute below.

```
auck$loc_mean <- apply(w, 1, function(i) mean(auck$TB[i]))
plot(auck, "loc_mean", main="Local mean", col=grays)
```



The local Moran *Ii* has not been implemented in `terra`, but we can use `spdep` (a package that if focussed on these type of statistics and on spatial regression).

```
library(spdep)
sfauck <- sf::st_as_sf(auck)
wr <- poly2nb(sfauck, row.names=sfauck$Id, queen=FALSE)
lstw <- nb2listw(wr, style="B")
auck$Ii <- localmoran(auck$TB, lstw)
## Warning: [[[<-,SpatVector] only using the first column
plot(auck, "Ii", main="Local Moran", col=grays)
```



In the above I followed the book by useing the raw count data. Howwever, it would be more approproate to use disease density instead. You can compute that like this:

```
auck$TBdens <- 10000 * auck$TB / expanse(auck)
```

The 10000 is just to avoid very small numbers.

7.3 Geographically weighted regression

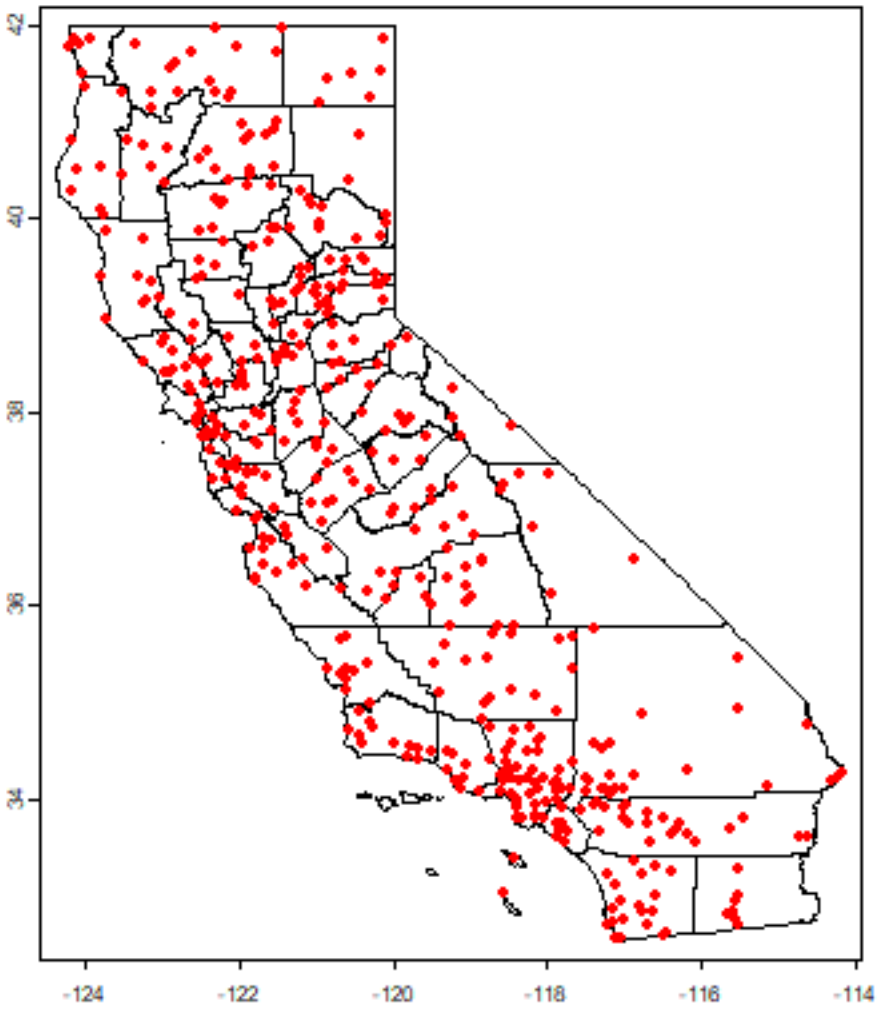
Here is an example of GWR with California precipitation data.

```
cts <- spat_data("counties")
p <- spat_data("precipitation")
head(p)
##      ID                  NAME   LAT    LONG ALT  JAN FEB MAR APR MAY JUN JUL
## 1 ID741         DEATH VALLEY 36.47 -116.87 -59  7.4 9.5 7.5 3.4 1.7 1.0 3.7
```

(continues on next page)

(continued from previous page)

```
## 2 ID743  THERMAL/FAA AIRPORT 33.63 -116.17 -34  9.2 6.9 7.9 1.8 1.6 0.4 1.9
## 3 ID744          BRAWLEY 2SW 32.96 -115.55 -31 11.3 8.3 7.6 2.0 0.8 0.1 1.9
## 4 ID753 IMPERIAL/FAA AIRPORT 32.83 -115.57 -18 10.6 7.0 6.1 2.5 0.2 0.0 2.4
## 5 ID754               NILAND 33.28 -115.51 -18  9.0 8.0 9.0 3.0 0.0 1.0 8.0
## 6 ID758        EL CENTRO/NAF 32.82 -115.67 -13  9.8 1.6 3.7 3.0 0.4 0.0 3.0
##    AUG SEP OCT NOV DEC
## 1  2.8 4.3 2.2 4.7 3.9
## 2  3.4 5.3 2.0 6.3 5.5
## 3  9.2 6.5 5.0 4.8 9.7
## 4  2.6 8.3 5.4 7.7 7.3
## 5  9.0 7.0 8.0 7.0 9.0
## 6 10.8 0.2 0.0 3.3 1.4

plot(cts)
points(p[,c("LONG", "LAT")], col="red", pch=20)
```



Compute annual average precipitation

```
p$pan <- rowSums(p[,6:17])
```

Global regression model

```
m <- lm(pan ~ ALT, data=p)
m
##
## Call:
## lm(formula = pan ~ ALT, data = p)
##
## Coefficients:
## (Intercept)          ALT
##      523.60         0.17
```

Create `sf` objects with a planar crs.

```
alb <- "+proj=aea +lat_1=34 +lat_2=40.5 +lat_0=0 +lon_0=-120 +x_0=0 +y_0=-4000000␣
↪+datum=WGS84"
sp <- vect(p, c("LONG", "LAT"), crs="+proj=longlat +datum=NAD83")
sp <- terra::project(sp, alb)
spsf <- sf::st_as_sf(sp)
vctst <- terra::project(cts, alb)
ctst <- sf::st_as_sf(vctst)
```

Get the optimal bandwidth

```
library( spgwr )
## Loading required package: sp
## NOTE: This package does not constitute approval of GWR
## as a method of spatial analysis; see example(gwr)
bw <- gwr.sel(pan ~ ALT, crds(sp), data=spsf)
## Bandwidth: 526221.2 CV score: 64886877
## Bandwidth: 850593.7 CV score: 74209069
## Bandwidth: 325747.9 CV score: 54001111
## Bandwidth: 201848.7 CV score: 44611207
## Bandwidth: 125274.7 CV score: 35746317
## Bandwidth: 77949.4 CV score: 29181734
## Bandwidth: 48700.75 CV score: 22737206
## Bandwidth: 30624.09 CV score: 17457182
## Bandwidth: 19452.1 CV score: 15163453
## Bandwidth: 12547.43 CV score: 19452224
## Bandwidth: 22792.75 CV score: 15513008
## Bandwidth: 17052.64 CV score: 15709986
## Bandwidth: 20218.98 CV score: 15167455
## Bandwidth: 19767.99 CV score: 15156930
## Bandwidth: 19790.05 CV score: 15156924
## Bandwidth: 19781.39 CV score: 15156920
## Bandwidth: 19781.48 CV score: 15156920
## Bandwidth: 19781.47 CV score: 15156920
## Bandwidth: 19781.47 CV score: 15156920
## Bandwidth: 19781.47 CV score: 15156920
## Bandwidth: 19781.47 CV score: 15156920
```

(continues on next page)

(continued from previous page)

```
bw
## [1] 19781.47
```

Create a regular set of points to estimate parameters for.

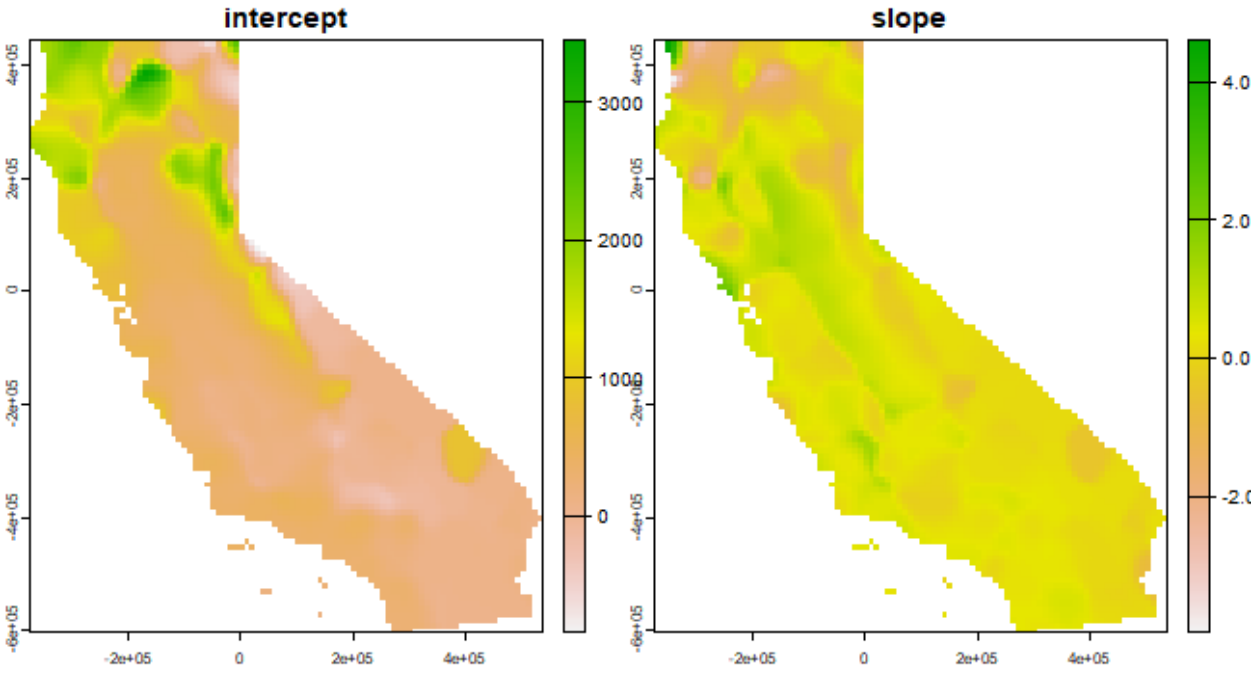
```
r <- rast(vctst, res=10000)
r <- rasterize(vect(ctst), r)
newpts <- geom(as.points(r))[, c("x", "y")]
```

Run the gwr function

```
g <- gwr(pan ~ ALT, crds(sp), data=spsf, bandwidth=bw, fit.points=newpts[, 1:2])
g
## Call:
## gwr(formula = pan ~ ALT, data = spsf, coords = crds(sp), bandwidth = bw,
##     fit.points = newpts[, 1:2])
## Kernel function: gwr.Gauss
## Fixed bandwidth: 19781.47
## Fit points: 4090
## Summary of GWR coefficient estimates at fit points:
##                       Min.     1st Qu.      Median     3rd Qu.      Max.
## X.Intercept. -846.293016   77.986323  328.578941  729.597303 3452.1971
## ALT            -3.961744    0.034145    0.201568    0.418714    4.6022
```

Link the results back to the raster.

```
slope <- intercept <- r
slope[!is.na(slope)] <- g$SDF$ALT
intercept[!is.na(intercept)] <- g$SDF$"(Intercept)"
s <- c(intercept, slope)
names(s) <- c("intercept", "slope")
plot(s)
```



See this page for a more detailed example of geographically weighted regression.

CHAPTER

EIGHT

FIELDS

8.1 Introduction

This handout accompanies Chapter 9 in O'Sullivan and Unwin (2010).

Here is how you can set up and use the continuous function on page 246.

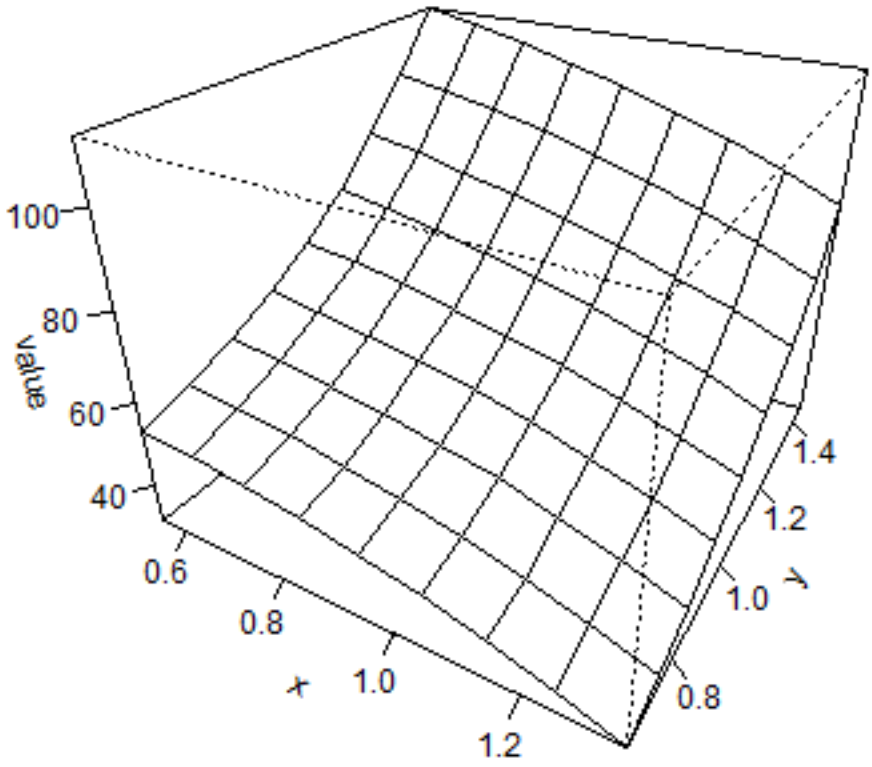
```
z <- function(x, y) { -12 * x^3 + 10 * x^2 * y - 14 * x * y^2 + 25 * y^3 + 50 }
z(.5, 0.8)
## [1] 58.82
```

Function `zf` adds some complexity to make it usable in the 'interpolate' function below.

```
zf <- function(model, xy) {
   x <- xy[,1]
   y <- xy[,2]
   z(x, y)
}
```

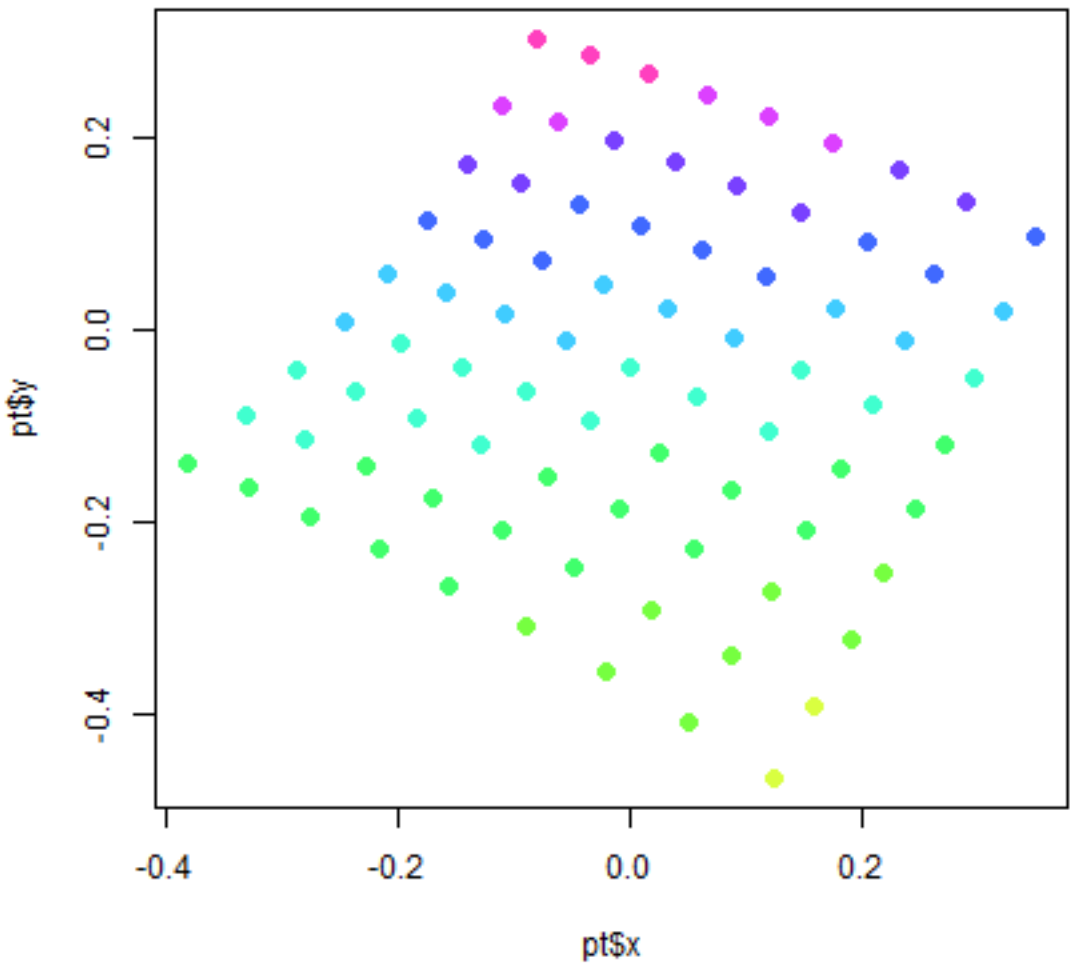
Now use it

```
library(terra)
## terra 1.7.62
r <- rast(xmin=0.5, xmax=1.4, ymin=0.6, ymax=1.5, ncol=9, nrow=9, crs="")
z <- interpolate(r, model=NULL, fun=zf)
names(z) <- 'z'

vt <- persp(z, theta=30, phi=30, ticktype='detailed', expand=.8)
```



Note that persp returned something *invisibly* (it won't be printed when not captured as a variable, `vt`, in this case), the 3D transformation matrix that we use later. This is not uncommon in R. For example `hist` and `barplot` have similar behaviour.

```
pts <- as.data.frame(z, xy=TRUE)
pt <- trans3d(pts[,1], pts[,2], pts[,3], vt)
plot(pt, col=rainbow(9, .75, start=.2)[round(pts[,3]/10)-2], pch=20, cex=2)
```



For a more interactive experience, try:

```
library(rasterVis)
library(rgl)
# this opens a new window
plot3D(raster::raster(z), zfac=5)
```

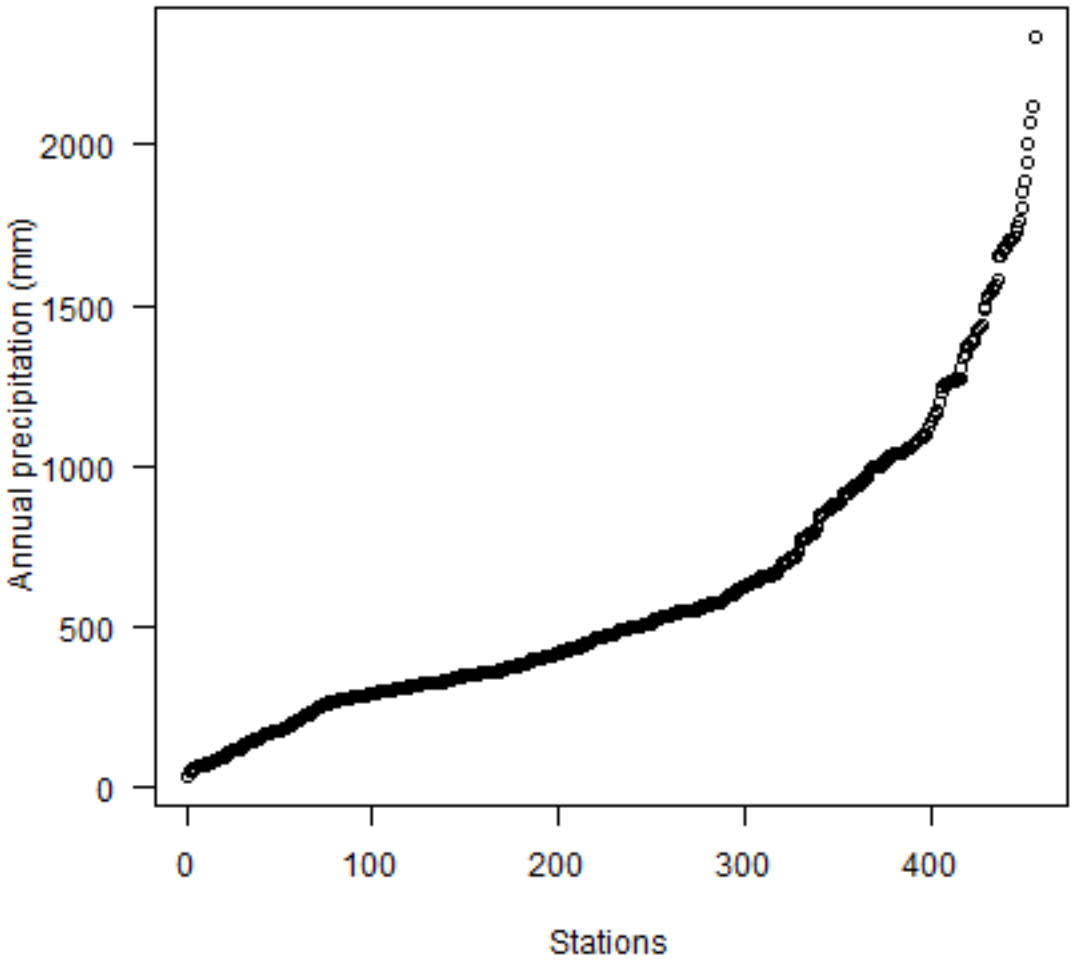
We will be working with temperature data for California. You can download the climate data used in the examples.

```
library(rspat)
d <- spat_data("precipitation")
head(d)
##      ID                  NAME   LAT    LONG ALT  JAN FEB MAR APR MAY JUN JUL
## 1 ID741         DEATH VALLEY 36.47 -116.87 -59  7.4 9.5 7.5 3.4 1.7 1.0 3.7
## 2 ID743  THERMAL/FAA AIRPORT 33.63 -116.17 -34  9.2 6.9 7.9 1.8 1.6 0.4 1.9
## 3 ID744           BRAWLEY 2SW 32.96 -115.55 -31 11.3 8.3 7.6 2.0 0.8 0.1 1.9
## 4 ID753 IMPERIAL/FAA AIRPORT 32.83 -115.57 -18 10.6 7.0 6.1 2.5 0.2 0.0 2.4
```

(continues on next page)

(continued from previous page)

```
## 5 ID754               NILAND 33.28 -115.51 -18  9.0 8.0 9.0 3.0 0.0 1.0 8.0
## 6 ID758        EL CENTRO/NAF 32.82 -115.67 -13  9.8 1.6 3.7 3.0 0.4 0.0 3.0
##    AUG SEP OCT NOV DEC
## 1  2.8 4.3 2.2 4.7 3.9
## 2  3.4 5.3 2.0 6.3 5.5
## 3  9.2 6.5 5.0 4.8 9.7
## 4  2.6 8.3 5.4 7.7 7.3
## 5  9.0 7.0 8.0 7.0 9.0
## 6 10.8 0.2 0.0 3.3 1.4
d$prec <- rowSums(d[, c(6:17)])
plot(sort(d$prec), ylab='Annual precipitation (mm)', las=1, xlab='Stations')
```

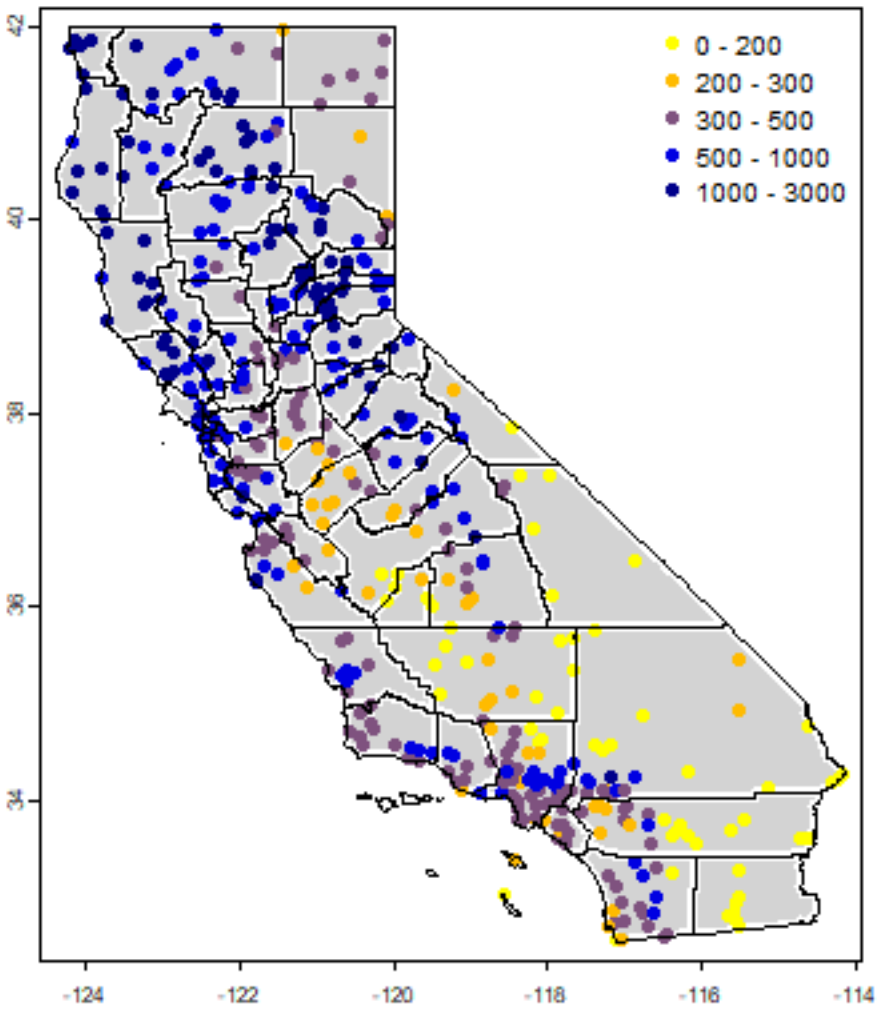



```
dsp <- vect(d, c("LONG", "LAT"), crs= "+proj=longlat +datum=WGS84")
CA <- spat_data("counties")
```

(continues on next page)

(continued from previous page)

```
cuts <- c(0,200,300,500,1000,3000)
blues <- colorRampPalette(c('yellow', 'orange', 'blue', 'dark blue'))
plot(CA, col="light gray", lwd=4, border="white")
plot(dsp, "prec", breaks=cuts, col=blues(5), pch=20, cex=1.5, add=TRUE, plg=list(x=
↪"topright"))
lines(CA, lwd=1.5)
```



CHAPTER

NINE

KRIGING

9.1 Alberta Rainfall

Recreating Figures 10.2, 10.13 & 10.14 in O'Sullivan and Unwin (2010).

We need the `rspat` packge to get the data we will use.

```
if (!require("rspat")) remotes::install_github('rspatial/rspat')
## Loading required package: rspat
## Loading required package: terra
## terra 1.7.62
```

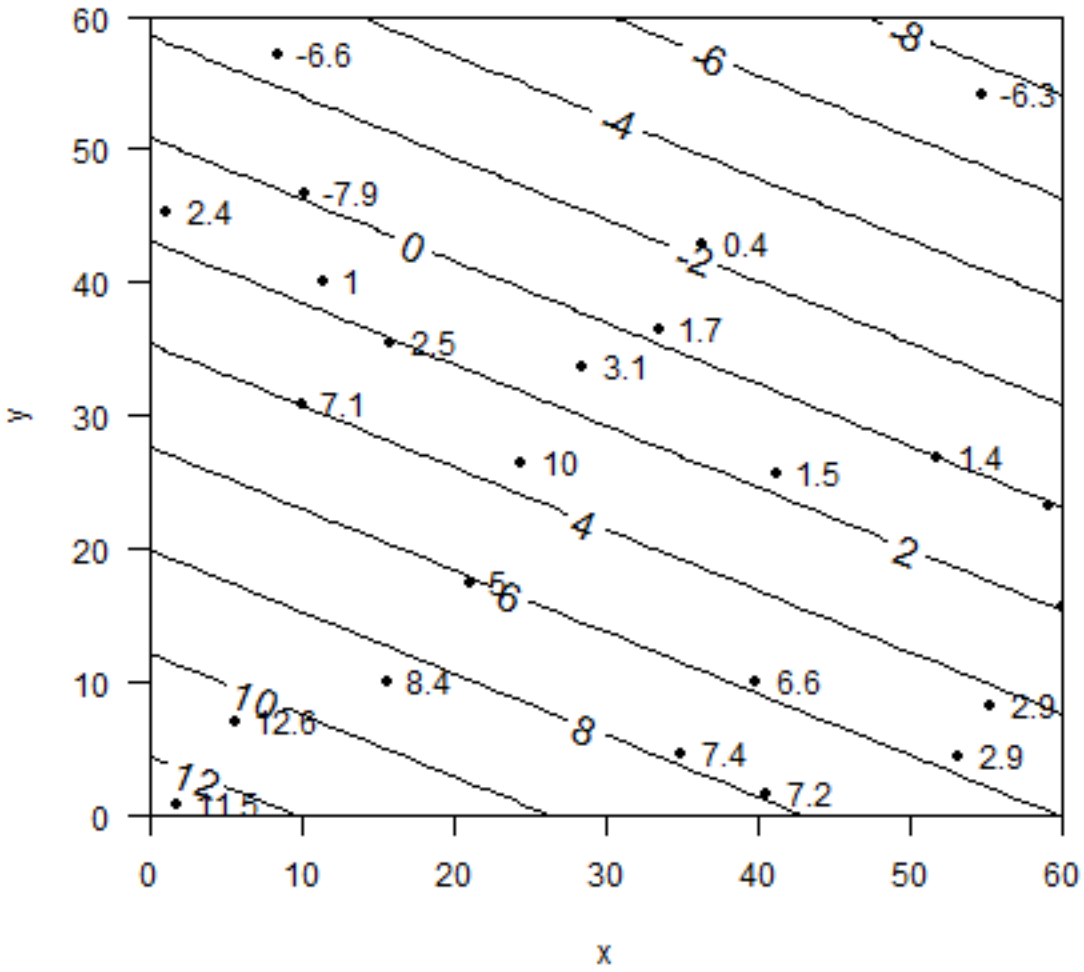
Figure 10.2

```
library(rspat)
a <- spat_data('alberta')

m <- lm(z ~ x + y, data=a)
summary(m)
##
## Call:
## lm(formula = z ~ x + y, data = a)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -7.763 -1.143 -0.081  1.520  6.600
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 13.16438    1.55646   8.458 3.34e-08 ***
## x           -0.11983    0.03163  -3.788  0.00108 **
## y           -0.25866    0.03584  -7.216 4.13e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.889 on 21 degrees of freedom
## Multiple R-squared:  0.7315, Adjusted R-squared:  0.7059
## F-statistic: 28.61 on 2 and 21 DF,  p-value: 1.009e-06
plot(a[,2:3], xlim=c(0,60), ylim=c(0,60), las=1, pch=20, yaxs="i", xaxs="i")
text(a[,2:3], labels=a$z, pos=4)
```

(continues on next page)

(continued from previous page)

```
# make the contour lines
x <- seq(0, 60, 1)
y <- seq(0, 60, 1)
# all combinations of x and y
xy <- data.frame(expand.grid(x=x,y=y))
z <- predict(m, xy)
z <- matrix(z, 61, 61)
contour(x, y, z, add=TRUE, labcex=1.25)
```



On to distances. First get a distance matrix for locations

```
library(terra)
m <- as.matrix(a[, c("x", "y")])
dp <- as.matrix(distance(m, lonlat=FALSE))
dim(a)
```

(continues on next page)

(continued from previous page)

```
## [1] 24  4
dim(dp)
## [1] 24 24
dp[1:5, 1:5]
##           1         2         3        4         5
## 1  0.000000  7.409453 44.504045 56.68554 46.662297
## 2  7.409453  0.000000 38.464139 50.07285 39.854862
## 3 44.504045 38.464139  0.000000 13.82317  9.108238
## 4 56.685536 50.072847 13.823169  0.00000 10.554620
## 5 46.662297 39.854862  9.108238 10.55462  0.000000
diag(dp) <- NA
```

Now the distance matrix for the values observated at the locations. 'dist' makes a symmetrical distance matrix that includes each pair only once. The distance function used above returns the distance between each pair twice. To illustrate this:

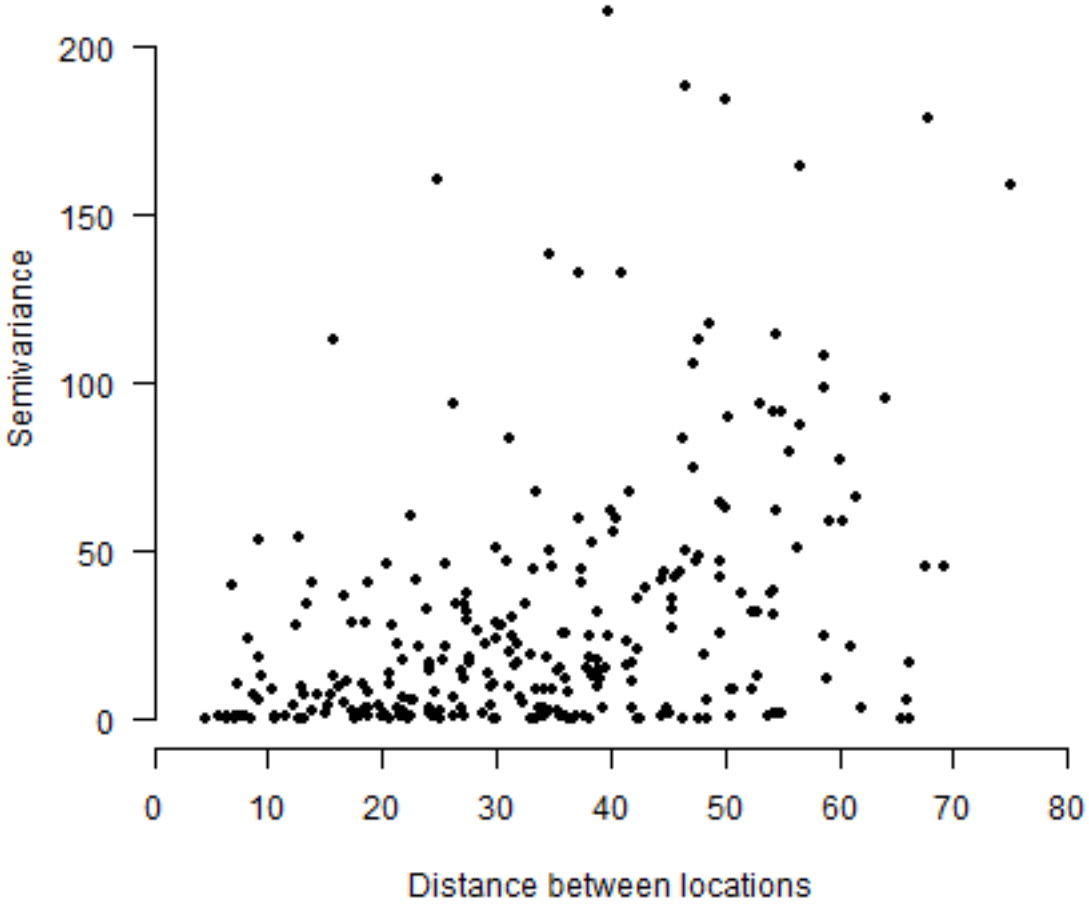
```
dist(a$z[1:3])
##     1   2
## 2 1.1
## 3 9.1 10.2
dz <- dist(a$z)
```

We can transform matrix dp to a distance matrix like this

```
dp <- as.dist(dp)
```

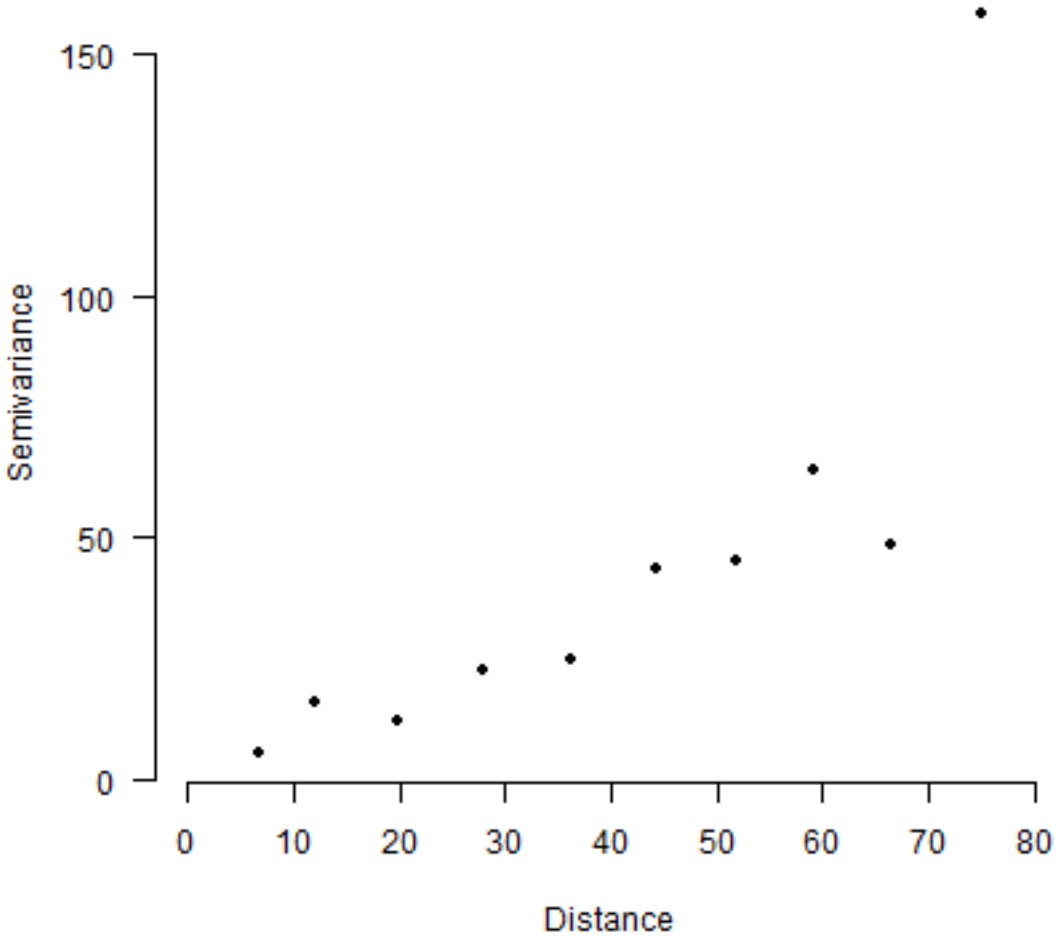
Plot a point cloud of spatial distance against the semivariance (Figure 10.13).

```
# semivariance
semivar <- dz^2 / 2

plot(dp, semivar, xlim=c(0, 80), ylim=c(0,220), xlab=c('Distance between locations'),
     ylab=c('Semivariance'), pch=20, axes=FALSE, xaxs="i")
axis(1, at=seq(0,80,10))
axis(2, las=1)
```



And plotting semivariance in bins (and note the "drift").

```
# choose a bin width (in spatial distance)
binwidth <- 8
# assign a lag (bin number) to each record
lag <- floor(dp/binwidth) + 1
# average value for each lag
lsv <- tapply(semivar, lag, mean)
# compute the average distance for each lag
dlag <- tapply(dp, lag, mean)

plot(dlag, lsv, pch=20, axes=FALSE, xlab='Distance', ylab='Semivariance', xlim=c(0,80))
axis(1, at=seq(0,80,10))
axis(2, las=1)
```



Now continue with the interpolation chapter of the "Spatial Data Analysis" section.

CHAPTER

TEN

MAP OVERLAY

10.1 Introduction

This document shows some example *R* code to do "overlays" and associated spatial data manipulation to accompany Chapter 11 in O'Sullivan and Unwin (2010). You have already seen many of this type of data manipulation in previous labs. And we have done perhaps more advanced things using regression type models (including LDA and RandomForest). This chapter is very much a review of what you have already seen: basic spatial data operations with *R*.

10.1.1 Get the data

You can get the data for this tutorial with the "rspat" package that you can install with the line below.

```
if (!require("rspat")) remotes::install_github('rspatial/rspat')
## Loading required package: rspat
## Loading required package: terra
## terra 1.7.62
```

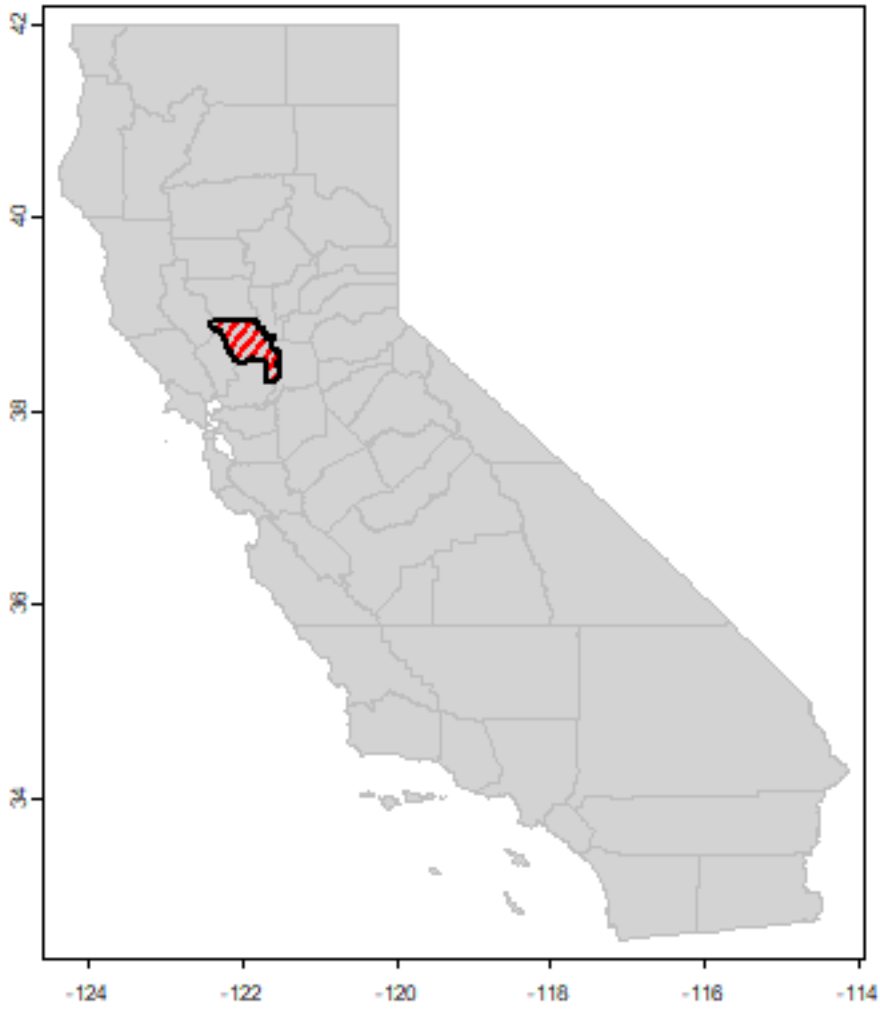
10.2 Selection by attribute

By now, you are well aware that in *R*, polygons and their attributes can be represented by a `SpatVector`. Here we use a `SpatVector` of California counties.

```
library(terra)
library(rspat)
counties <- spat_data('counties')
```

Selecting rows (geometries and their attributes) of a `SpatVector` is similar to selecting rows from a `data.frame`. For example, to select Yolo county by its name:

```
yolo <- counties[counties$NAME == 'Yolo', ]
plot(counties, col='light gray', border='gray')
plot(yolo, add=TRUE, density=20, lwd=2, col='red')
```



You can interactively select counties this way by clicking on the map of the counties:

```
plot(counties)
```

```
s <- sel(counties)
```

10.3 Intersection and buffer

I want to get the railroads in the city of Davis by sub-setting the railroads in Yolo county. First read the data, and do an important sanity check: are the coordinate reference systems (crs) for the two datasets the same?

```
rail <- spat_data("yolo-rail")
rail
##  class       : SpatVector
##  geometry    : lines
##  dimensions  : 4, 1  (geometries, attributes)
```

(continues on next page)

(continued from previous page)

```
##  extent      : -122.0208, -121.5089, 38.31305, 38.9255  (xmin, xmax, ymin, ymax)
##  coord. ref. : +proj=longlat +datum=WGS84 +no_defs
##  names       :               FULLNAME
##  type        :                  <chr>
##  values      :           Abandoned RR
##                California Northern RR
##                     Union Pacific RR
class(rail)
## [1] "SpatVector"
## attr(,"package")
## [1] "terra"
city <- spat_data("city")

crs(yolo, TRUE)
## [1] "+proj=longlat +datum=WGS84 +no_defs"
crs(rail, TRUE)
## [1] "+proj=longlat +datum=WGS84 +no_defs"
crs(city, TRUE)
## [1] "+proj=lcc +lat_0=37.6666666666667 +lon_0=-122 +lat_1=38.3333333333333 +lat_2=39.
↪8333333333333 +x_0=2000000 +y_0=500000.000000001 +datum=WGS84 +units=us-ft +no_defs"
```

Ay, we are dealing with two different coordinate reference systems (projections)! Let's settle for yet another one: Teale Albers. This is a short name for the "Albers Equal Area projection with parameters suitable for California". This particular set of parameters was used by an California State organization called the Teale Data Center, hence the name.

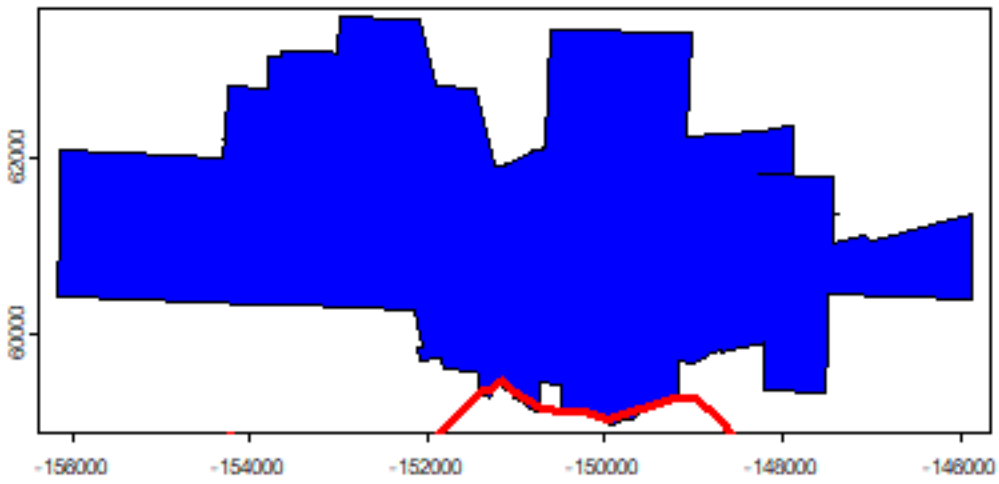
```
TA <- "+proj=aea +lat_1=34 +lat_2=40.5 +lat_0=0 +lon_0=-120 +x_0=0 +y_0=-4000000␣
↪+datum=WGS84 +units=m"
countiesTA <- project(counties, TA)
yoloTA <- project(yolo, TA)
railTA <- project(rail, TA)
cityTA <- project(city, TA)
```

Another check, let's see what county Davis is in, using two approaches. In the first one we get the centroid of Davis and do a point-in-polygon query.

```
davis <- centroids(cityTA)
i <- which(relate(davis, countiesTA, "intersects"))
countiesTA$NAME[i]
## [1] "Yolo"
```

An alternative approach is to intersect the two polygon SpatVectors

```
i <- intersect(cityTA, countiesTA)
i$area <- expanse(i)
as.data.frame(i)
##   STATE COUNTY   NAME LSAD LSAD_TRANS        area
## 1    06    095 Solano   06     County    78791.82
## 2    06    113   Yolo   06     County 25633905.73

plot(cityTA, col='blue')
plot(yoloTA, add=TRUE, border='red', lwd=3)
```



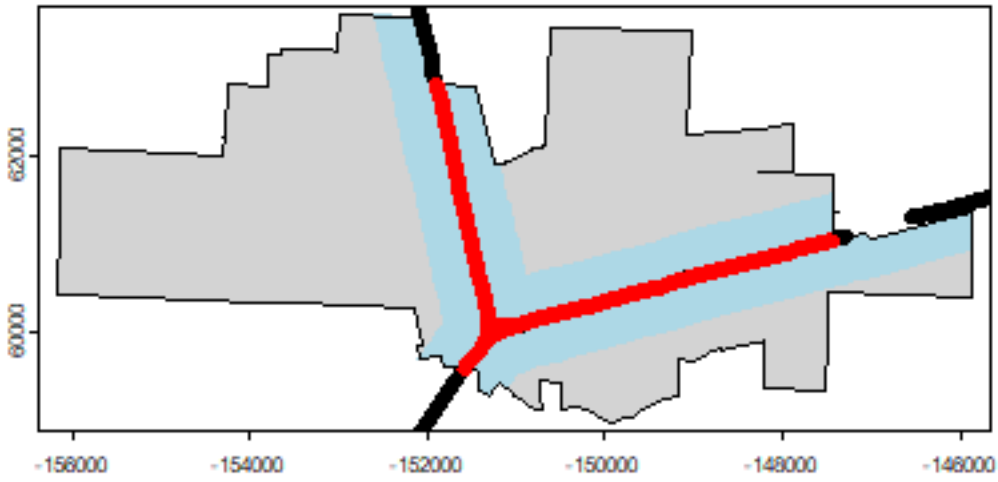
Our data suggest that we have a little sliver of Davis inside of Solano county (the county to the south of Yolo) — but that is probably not correct. At least it would seem likely that the boundaries may be the same in reality.

Otherwise, everything looks OK. Now we can intersect rail and city, and make a buffer.

```
davis_rail <- intersect(railTA, cityTA)
```

Compute a 500 meter buffer around railroad inside Davis:

```
buf <- buffer(railTA, width=500)
buf <- aggregate(buf)
rail_buf <- intersect(buf, cityTA)

plot(cityTA, col='light gray')
plot(rail_buf, add=TRUE, col='light blue', border='light blue')
plot(railTA, add=TRUE, lty=2, lwd=6)
plot(cityTA, add=TRUE)
plot(davis_rail, add=TRUE, col='red', lwd=6)
```



What is the percentage of the area of the city of Davis that is within 500 m of a railroad?

```
round(100 * expanse(rail_buf) / expanse(cityTA))
## [1] 31
```

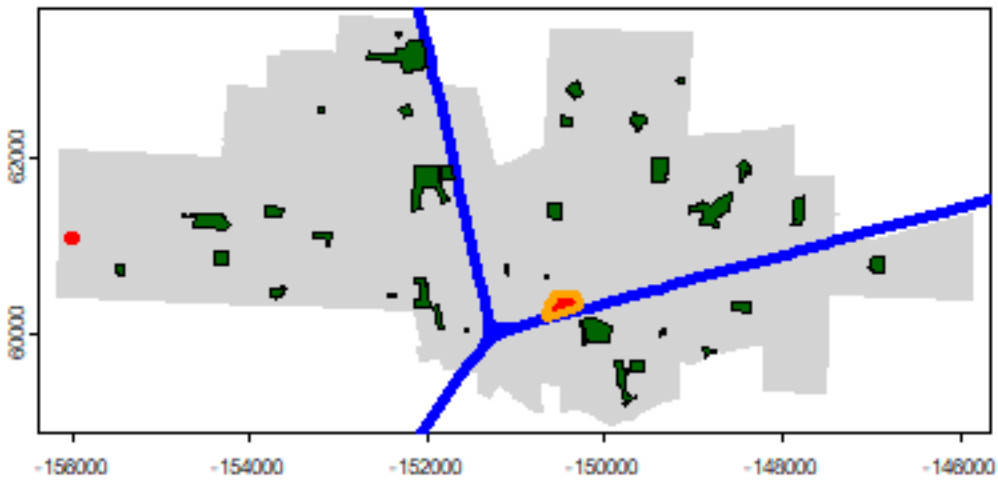
10.4 Proximity

Which park in Davis is furthest, and which is closest to the railroad? First get the parks data.

```
parks <- spat_data("parks")
crs(parks, TRUE)
## [1] "+proj=lcc +lat_0=37.6666666666667 +lon_0=-122 +lat_1=38.3333333333333 +lat_2=39.
↪8333333333333 +x_0=2000000 +y_0=500000.0000000001 +datum=WGS84 +units=us-ft +no_defs"
parksTA <- project(parks, TA)
```

Now plot the parks that are the furthest and the nearest from a railroad.

```
plot(cityTA, col="light gray", border="light gray")
plot(railTA, add=TRUE, col="blue", lwd=4)
plot(parksTA, col="dark green", add=TRUE)

d <- distance(parksTA, railTA)
dmin <- apply(d, 1, min)
parksTA$railDist <- dmin

i <- which.max(dmin)
as.data.frame(parksTA)[i,]
##               PARK     PARKTYPE            ADDRESS railDist
## 26 Whaleback Park NEIGHBORHOOD 1011 MARINA CIRCLE 4330.938
plot(parksTA[i, ], add=TRUE, col="red", lwd=3, border="red")

j <- which.min(dmin)
as.data.frame(parksTA)[j,]
##                     PARK     PARKTYPE         ADDRESS railDist
## 14 Toad Hollow Dog Park NEIGHBORHOOD 1919 2ND STREET 23.83512
plot(parksTA[j, ], add=TRUE, col="red", lwd=3, border="orange")
```



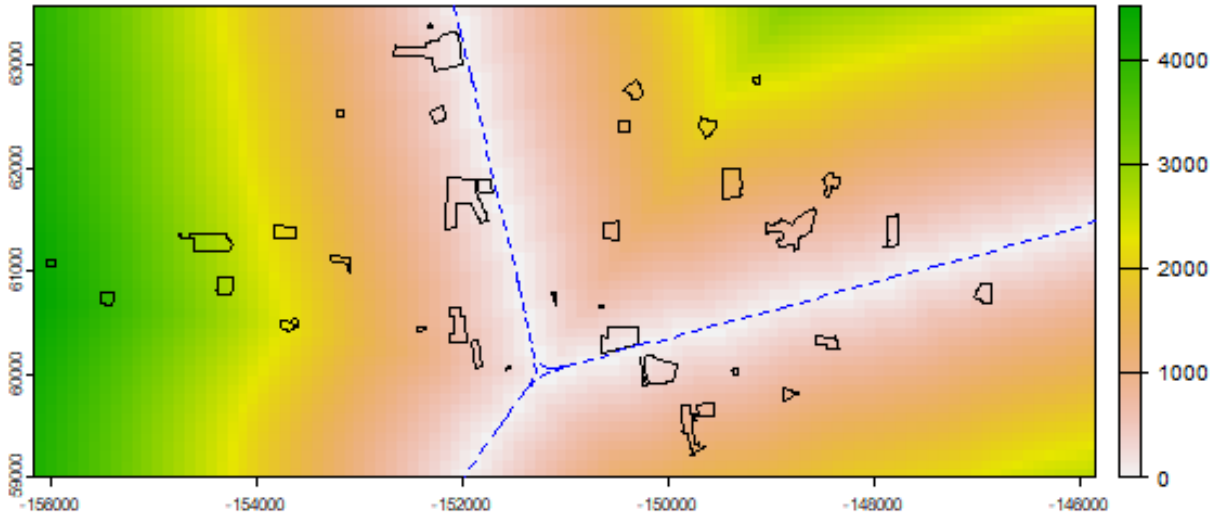
Another way to approach this is to first create a raster with distance to the railroad values. Here we compute the average distance to any place inside the park. You could also compute the distance to the border or the centroids of the parks.

```
# use cityTA to set the geographic extent
r <- rast(cityTA)

# arbitrary resolution
dim(r) <- c(50, 100)

# rasterize the railroad lines
r <- rasterize(railTA, r)

# compute distance
d <- distance(r)
names(d) <- "dist"

# extract distance values for polygons
```

(continues on next page)

(continued from previous page)

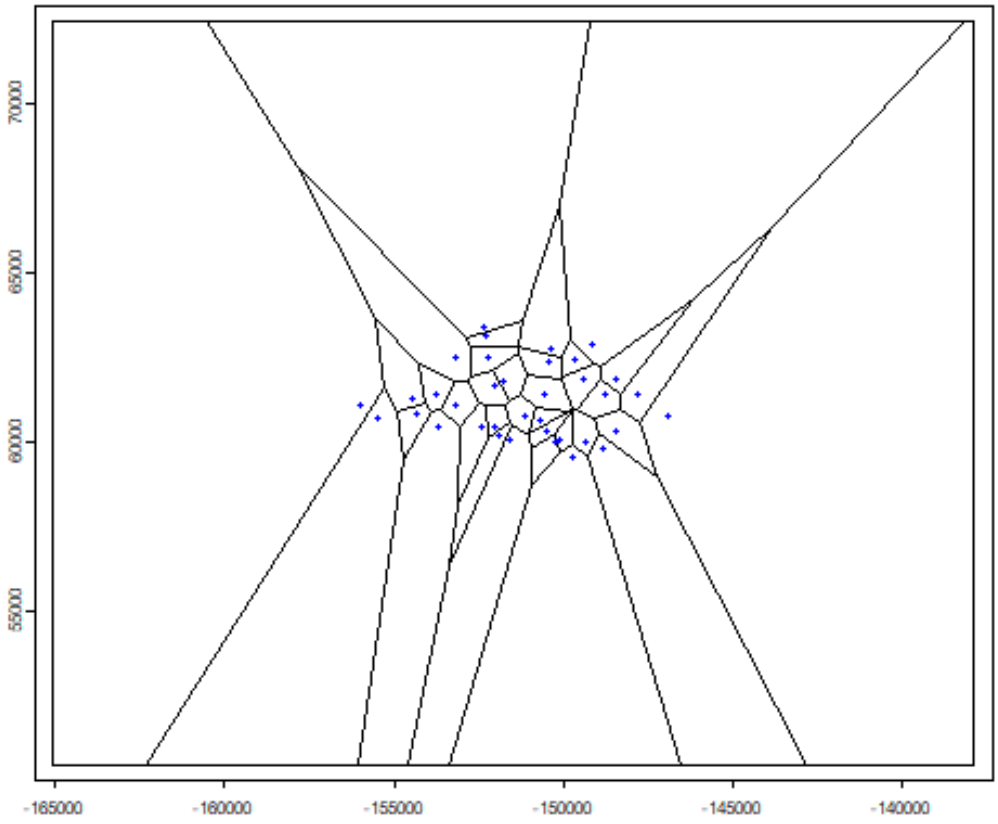
```
dp <- extract(d, parksTA, fun=mean, exact=TRUE)

dp <- data.frame(park=parksTA$PARK, distance=dp[,2])
dp <- dp[order(dp$distance), ]

plot(d)
plot(parksTA, add=TRUE)
plot(railTA, add=T, col="blue", lty=2)
```



10.4.1 Voronoi polygons

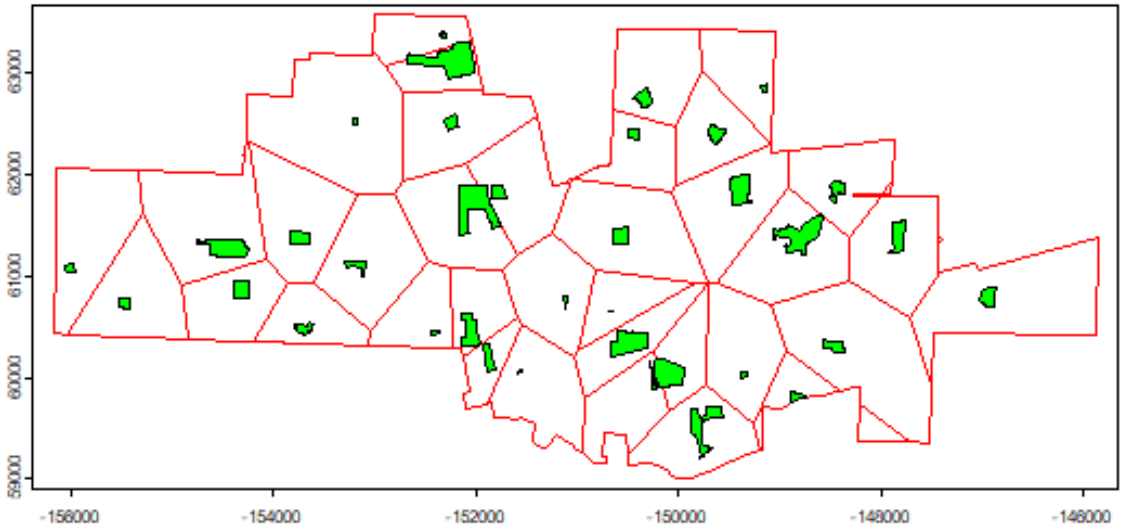
Here I compute Voronoi (or Thiessen) polygons for the centroids of the Davis parks. Each polygon shows the area that is closest to (the centroid of) a particular park.

```
centr <- centroids(parksTA)
v <- voronoi(centr)
plot(v)
points(centr, col="blue", pch=20)
```



To keep the polygons within Davis.

```
vc <- intersect(v, cityTA)
plot(vc, border="red")
plot(parksTA, add=T, col="green")
```



10.5 Raster data

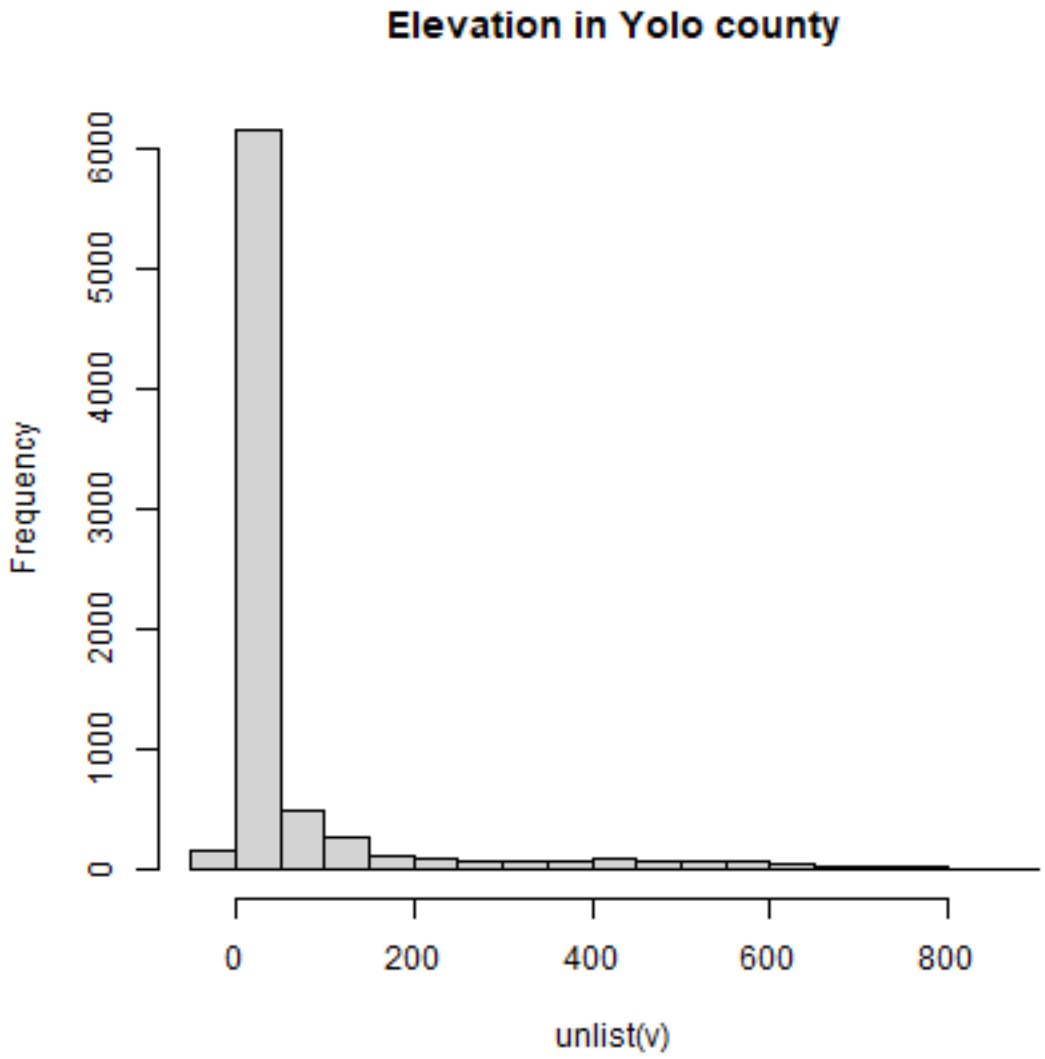
raster data can be read with the `rast` method. See the terra package for more details

```
alt <- spat_data("elevation")
alt
## class       : SpatRaster
## dimensions  : 239, 474, 1  (nrow, ncol, nlyr)
## resolution  : 0.008333333, 0.008333333  (x, y)
## extent      : -123.95, -120, 37.3, 39.29167  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs
## source(s)   : memory
## name        : lyr.1
## min value   :   -30
## max value   :  2963
```

10.5.1 Query

Now extract elevation data for Yolo county.

```
v <- extract(alt, yolo)
hist(unlist(v), main="Elevation in Yolo county")
```



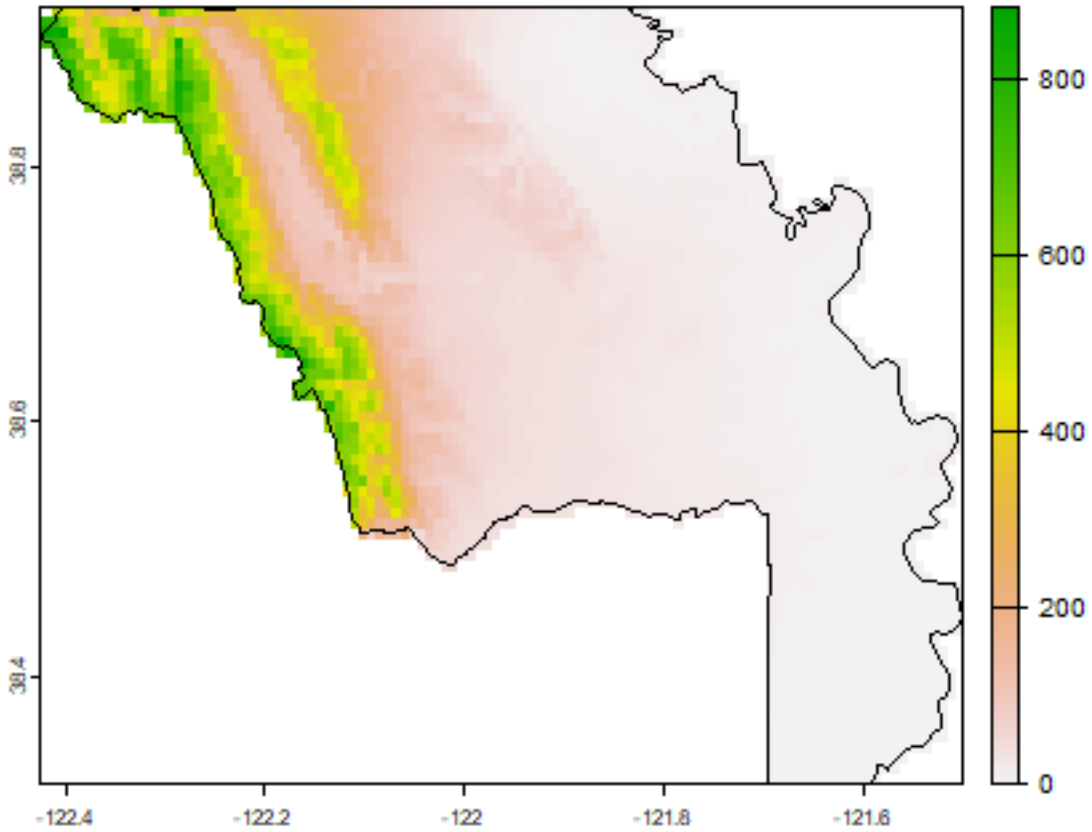
Another approach:

```
# cut out a rectangle (extent) of Yolo
yalt <- crop(alt, yolo)
# "mask" out the values outside Yolo
ymask <- mask(yalt, yolo)

# summary of the raster cell values
summary(ymask)
##      lyr.1
```

(continues on next page)

(continued from previous page)

```
##  Min.    : -1.0
##  1st Qu.:  6.0
##  Median : 28.0
##  Mean   :111.3
##  3rd Qu.:110.0
##  Max.   :882.0
##  NA's   :3972

plot(ymask)
plot(yolo, add=T)
```



You can also get values (query) by clicking on the map (use `click(alt)`)

10.6 Exercise

We want to travel by train in Yolo county and we want to get as close as possible to a hilly area; whenever we get there we'll jump from the train. It turns out that the railroad tracks are not all connected, we will ignore that inconvenience.

Define "hilly" as some vague notion of a combination of high elevation and slope (in degrees). Use some of the functions you have seen above, as well as function 'distance' to create a plot of distance from the railroad against hillyness. Make a map to illustrate the result, showing where you get off the train, where you go to, and what the elevation and slope profile would be if you follow the shortest (as the crow flies) path.

Bonus: use the `costDist` function to find the least-cost path between these two points (assign a cost to slope, perhaps using Tobler's hiking function).

CHAPTER

ELEVEN

APPENDIX

This page accompanies the Appendix of O'Sullivan and Unwin (2010).

Add two matrices

```
A <- matrix(1:4, nrow=2, byrow=TRUE)
B <- matrix(5:8, nrow=2, byrow=TRUE)
A + B
##      [,1] [,2]
## [1,]    6    8
## [2,]   10   12
```

Matrix multiplication

```
A <- matrix(c(1,-4,-2,5,3,-6), nrow=2)
B <- matrix(c(6,4,2,-5,-3,-1), nrow=3)
A %*% B
##      [,1] [,2]
## [1,]    4   -2
## [2,]  -16   11
B %*% A
##      [,1] [,2] [,3]
## [1,]   26  -37   48
## [2,]   16  -23   30
## [3,]    6   -9   12
```

Matrix transposition

```
A <- matrix(1:6, nrow=2, byrow=TRUE)
A
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
t(A)
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Identity matrix

```
I <- matrix(0, ncol=2, nrow=2)
diag(I) <- 1
```

(continues on next page)

(continued from previous page)

```
I
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1

I <- matrix(0, ncol=5, nrow=5)
diag(I) <- 1
I
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

Finding the inverse matrix

```
A <- matrix(1:4, nrow=2, byrow=TRUE)
Inv <- solve(A)
Inv
##      [,1] [,2]
## [1,] -2.0  1.0
## [2,]  1.5 -0.5

AA <- A %*% Inv
AA
##      [,1]         [,2]
## [1,]    1 1.110223e-16
## [2,]    0 1.000000e+00
round(AA, 10)
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

inv(AB) == inv(A) * inv(B)

```
A <- matrix(1:4, nrow=2, byrow=TRUE)
B <- matrix(4:1, nrow=2, byrow=TRUE)
AB <- A %*% B
solve(AB)
##       [,1]  [,2]
## [1,]  3.25 -1.25
## [2,] -5.00  2.00
# the same as
solve(B) %*% solve(A)
##       [,1]  [,2]
## [1,]  3.25 -1.25
## [2,] -5.00  2.00
```

Simulataneous equations

```
A <- matrix(c(3,2,4,-4), nrow=2)
b <- matrix(c(11, -6))
```

(continues on next page)

(continued from previous page)

```
solve(A) %*% b
##      [,1]
## [1,]    1
## [2,]    2
```

Rotation

```
A <- matrix(c(.6,-.8,.8,.6), nrow=2)
s <- matrix(c(3, 4))
As <- A %*% s
round(As, 10)
##      [,1]
## [1,]    5
## [2,]    0

S <- matrix(c(1,1,3,-2,0,5,-1,4,-2.5,-4), nrow=2)
AS <- A %*% S
AS
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  1.4  0.2    4  2.6 -4.7
## [2,] -0.2 -3.6    3  3.2 -0.4
```

The angle of rotation matrix `A` is

```
angle <- acos(A[1])
angle
## [1] 0.9272952
# in degrees
180*angle/ pi
## [1] 53.1301
```

See this page for more on rotation matrices.

Eigenvector and values

```
M <- matrix(c(3,2,4,-4), nrow=2)
eigen(M)
## eigen() decomposition
## $values
## [1] -5  4
##
## $vectors
##            [,1]      [,2]
## [1,] -0.4472136 0.9701425
## [2,]  0.8944272 0.2425356

M <- matrix(c(1,3,3,2), nrow=2)
eigen(M)
## eigen() decomposition
## $values
## [1]  4.541381 -1.541381
##
## $vectors
```

(continues on next page)

(continued from previous page)

```
##           [,1]       [,2]
## [1,] 0.6463749 -0.7630200
## [2,] 0.7630200  0.6463749
```