3

Statistical Properties of Spatially Autocorrelated Data

3.1 Introduction

Waldo Tobler made famous his "first law of geography," which states that "everything is related to everything else, but near things are more related than distant things" (Tobler, 1970, p. 236), but he was obviously not the first person to realize this. Awareness of the need to take into account the effect of temporal and spatial proximity when carrying out statistical analyses goes back at least as far as Student (1914). Nowadays, data that obey the first law of geography are said to be *spatially autocorrelated*. The prefix "auto" comes from the Greek word *αυτό*, meaning "self." Thus, to say a feature is spatially autocorrelated means etymologically that its attribute values are correlated with attribute values of the same feature at nearby locations. Different authors define the term *spatial autocorrelation* differently. Anselin and Bera (1998, p. 241) provide a concise verbal definition: "spatial autocorrelation can be loosely defined as the coincidence of value similarity with locational similarity." For example, one of the quantities recorded in the Weislander survey illustrated in Figure 1.1 is mean annual precipitation. Nearby locations would probably tend to have similar mean annual precipitation levels. Anselin and Bera (1998) also provide a more formal definition as follows: a nonzero spatial autocorrelation exists between attributes of a feature defined at locations i and j if the covariance between feature attribute values at those points is nonzero. If this covariance is positive (i.e., if data with attribute values above the mean tend to be near other data with values above the mean), then we say there is *positive* spatial autocorrelation; if the converse is true, then we say there is *negative* spatial autocorrelation. Positive autocorrelation is much more common in nature, but negative autocorrelation does exist, for example, in the case of conspecific allelopathy, the tendency of some plants to inhibit the nearby growth of other plants of the same species (Rice, 1984). Nevertheless, in this book whenever we use the term spatial autocorrelation we will mean *positive* spatial autocorrelation.

The areal data discussed in this book generally consist of sets of attribute values $Y(x, y)$ together with the coordinates x and y describing the location of each attribute value. The locations (x, y) are fixed and not random, but the attribute values Y are assumed to have a random component. Errors in measuring the location of the phenomenon are incorporated into the uncertainty in the value of the phenomenon at each location. The coordinates themselves are assumed to be measured without error. For example, Figure 1.3 shows the values of clay content in a farmer's field. Error in this case would represent an inaccurate measurement of clay content, but the locations are assumed to be measured accurately.

Consider now the process of modeling a data set such as that of Figure 1.3. A statistical model includes a *random variable*, that is, very roughly speaking, a quantity that is sampled and whose values are distributed according to some probability distribution. A more rigorous discussion of random variables is provided by Larsen and Marx (1986, p. 104). A random variable that is measured at a set of locations is called a *random field* (Besag, 1974, Cressie, 1991, p. 8). Any particular set of measurements of the random field (e.g., the set of clay content measurements illustrated in Figure 1.3), is called a *realization* of the random field (or of the random variable). The random variable may not actually be measurable at every point in the domain. For example, if we are measuring tree yield in an orchard, this quantity is not measurable at locations where there is no tree. The manner in which we deal with this situation will depend on the class of spatial data model (geostatistical or areal, see Section 1.2.1) that we employ in the analysis.

Each data record in every spatial data set is identified as having a location specified by an (x, y) coordinate pair or by a polygon. For the clay content data of Figure 1.3, for example, this location is the point at which the measurement is made. Figure 1.5 shows a spatial mosaic whose cells are polygons defined by land cover class. Observations were made in these cells of the presence or absence of the western yellow-billed cuckoo, with no more than one observation site in each cell. The observations were made by playing a recording of the bird's call and listening for a response. In this case, there is no natural choice for the precise location of the coordinate pair describing the observation site, and it is simply identified as being in the polygon.

3.2 Components of a Spatial Random Process

3.2.1 Spatial Trends in Data

It is often convenient to model a spatial random field as the sum of a collection of separate components, each with its own properties. Cressie (1991, p. 112) presents a model for spatial variability in which the data consist of the sum of four components with differing scales of variability. Burrough and McDonnell (1998, p. 134) describe a slightly simpler system that will suit our needs. In this system, the data are represented as consisting of the sum of three components. These are (1) a "structural," or "deterministic" component that may consist of a trend, of large scale variation, or both; (2) a spatially autocorrelated random process; and (3) an uncorrelated random variable representing uncorrelated random variation and measurement error. Displaying the (x, y) coordinates explicitly, one can write this as

$$Y(x, y) = T(x, y) + \eta(x, y) + \varepsilon(x, y). \tag{3.1}$$

Here $T(x, y)$ represents the deterministic trend, $\eta(x, y)$ represents the spatially autocorrelated random process, and $\varepsilon(x, y)$ represents the uncorrelated random variable. This idea is shown schematica1lly in Figures 3.1 and 3.2 (code is also included to create Figure 3.1 using `ggplot()`). If, for example, $Y(x, y)$ represents mean annual precipitation at the point (x, y), then we might imagine that there is some underlying, large-scale variability that may be modeled as the trend $T(x, y)$ (e.g., due to elevation, slope, and aspect). There are smaller scale components that, taken together, may be modeled as the autocorrelated

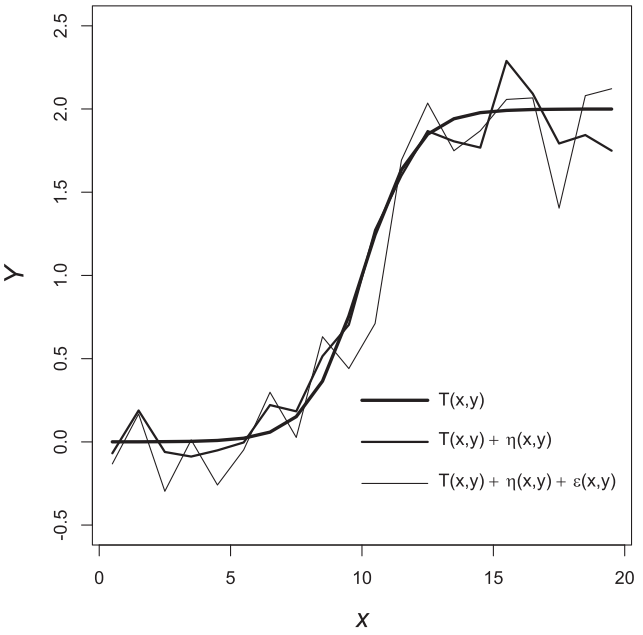

FIGURE 3.1
Plot of a cross section of the variable $Y(x, y)$ shown in perspective in Figure 3.2a.

random variable $\eta(x, y)$ (e.g., due to small scale topographic features and low altitude air movement), and there are still other random components that taken together may be modeled as uncorrelated variability and measurement error $\varepsilon(x, y)$ (Haining, 2003, p. 185). It is important to emphasize that in dealing with real data, any decomposition of the form of Equation 3.1 cannot be unique. To quote Cressie (1991, p. 114), "one person's deterministic trend may be another person's correlated error structure." Keep this in mind as you read the following discussion.

In order to separate a spatial data set into components, one must first estimate the trend term $T(x, y)$ and, if it is large enough to warrant attention, subtract it from the data. The trend is often estimated using either a linear regression model (Appendix A.2) or the *median polish* technique, although a generalized additive model can also be very effective (Dormann et al., 2007, see Exercise 9.3). To use a linear regression model, one fits the data with a regression function in which the predictor variables are functions of the coordinates x and y. The simplest such function is a first-degree model of the form

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 y_i + \varepsilon_i, \tag{3.2}$$

where Y_i is the measured value at the location (x_i, y_i), $i = 1, \ldots, n$. Other more complex models, either linear regression models with higher degree terms in the coordinates or nonlinear regression or generalized additive models, may also be used (Unwin, 1975). The parameters of the regression model may be estimated using ordinary or nonlinear least squares, depending on the model. One must exercise caution in interpreting the results of least squares with polynomial terms since the regression may be ill-conditioned (i.e., have numerical properties that lead to an inaccurate solution) (Haining, 2003, p. 326).

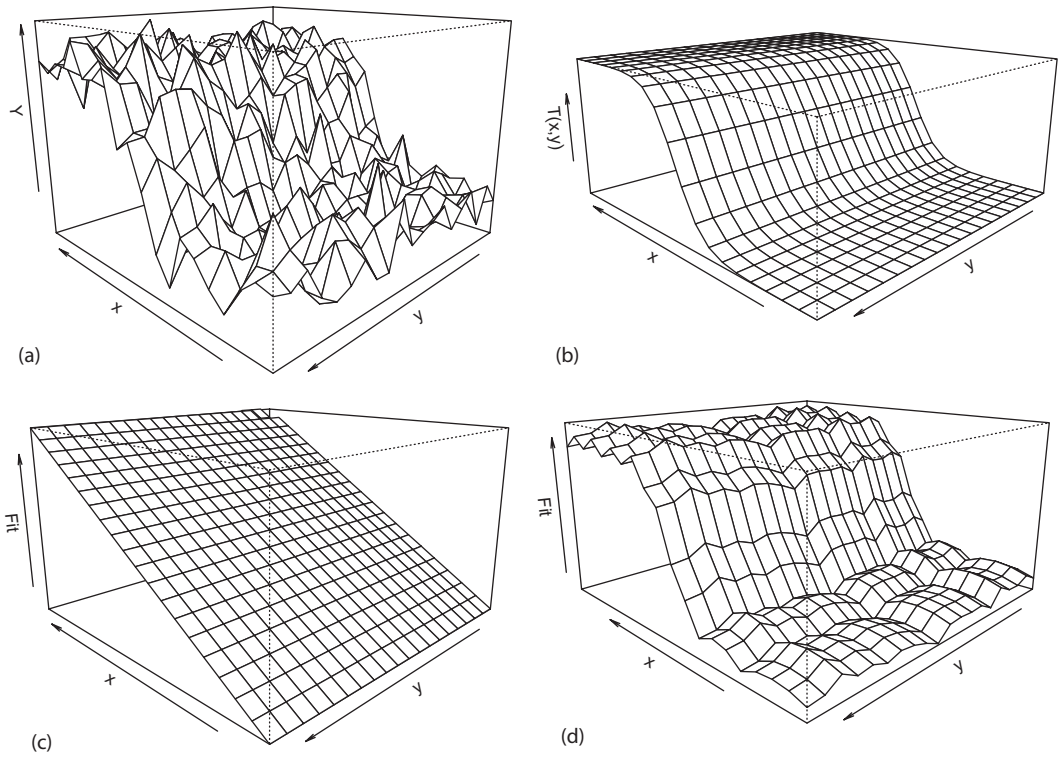

FIGURE 3.2
(a) Perspective plot of a random field $Y(x, y)$ made up of a trend surface, an autocorrelated random component, and an uncorrelated random component; (b) trend $T(x, y)$ of the random field; (c) linear regression estimate of $T(x, y)$; (d) median polish estimate of $T(x, y)$.

To illustrate the process of fitting a trend surface, we will employ a made-up example consisting of a random field $Y(x, y)$ satisfying Equation 3.1 with the following components. The deterministic component $T(x, y)$ is a logistic function of the form

$$T(x, y) = \frac{ae^{-cy}}{1 + e^{-cy}}. \tag{3.3}$$

where a has the value 2 and c has the value 1. Figure 3.2a shows a perspective plot of the random variable $Y(x, y)$ whose cross section is shown in Figure 3.1. Figure 3.2b shows the actual trend surface component $T(x, y)$ First, we estimate the trend $T(x, y)$ using linear regression with Equation 3.2. The R function most commonly used for ordinary linear regression is `lm()`. This function is discussed in Appendix A.2. The R package `spatial` (Venables and Ripley, 2002, p. 420) contains a function `surf.ls()` that can also be used to fit a trend surface by least squares, but for this simple case doing it directly with `lm()` is easier and more transparent. The values of $Y(x, y)$ are stored in a data frame called `Y.df`. The statements that create this data frame are in the code that accompanies this book. The code to compute the estimated trend and display a perspective plot on a 20 by 20 grid is the following.

```
> model.lin <- lm(Y ~ x + y, data = Y.df)
> coef(model.lin)
        (Intercept)           x           y
      -0.435929572 0.143930575 0.000212603
> trend.lin <- predict(model.lin)
> Y.lin <- matrix(trend.lin, nrow = 20)
> Fit <- 4 * Y.lin
> persp(x, y, Fit, theta = 225, phi = 15, scale = FALSE)
```

The first argument in `lm()` is the regression formula (see Appendix A.2), and the second is the source of the data. The estimated regression coefficients are $b_0 = -0.436$ $b_1 = 0.144$, and $b_3 = 0.0002$. The function `predict()` gives the predicted values of the linear model at the data points. The next two lines of code generate the trend surface shown in Figure 3.2c. The function `persp(x, y, Y, theta, phi, scale...)` displays a perspective plot of `Y` as a function of `x` and `y`. The arguments `theta` and `phi` give the azimuth and latitude from with the plot is viewed, and `scale` is a logical argument that, if true, allows each axis to be scaled independently. In lieu of this, we simply multiply the z component by four (obtained by trial and error) to exaggerate it.

The second approach to fitting a trend surface, median polish, is a nonparametric method, that is, one in which no assumptions are made about the distribution of the error terms, in the way that they are with linear regression. Median polish is an iterative procedure originally described by Tukey (1977) and discussed by Emerson and Hoaglin (1983) and Cressie (1991, p. 185). At each iteration, one alternately subtracts row and column medians from the data set. The R function that performs median polish is `medpolish()`. The code to carry out the operations is

```
> Y.trend <- matrix(Y.df$Y, nrow = 20)
> model.mp <- medpolish(Y.trend)
> Y.mp <- Y.trend - model.mp$residuals
> Fit <- 4 * Y.mp
> persp(x, y, Fit, theta = 225, phi = 15, scale = FALSE)
```

The median polish trend surface of the simulated data set is shown in Figure 3.2d. In this example, the median polish fit more accurately represents the trend surface, and because of the logistic shape of this surface, median polish would probably still be more accurate even if higher order powers of x and y were included in the regression in Equation 3.2.

The choice of method for trend estimation depends on the application, and, to some extent, on the trend surface. Median polish frequently provides a more accurate fit $\hat{T}(x, y)$ of the trend, which is useful for visualization and, if the trend surface is complex, subtracting this estimate from $Y(x, y)$ provides a more accurate prediction $\hat{\eta}(x, y) + \hat{\varepsilon}(x, y)$ of the random quantity represented as the sum $\eta(x, y) + \varepsilon(x, y)$ in Equation 3.1. On the other hand, the parameter values generated by fitting a regression model can provide useful summary information, such as whether the trend is stronger in the x or y direction, or whether it has a strong quadratic component or interaction, etc. The parameter values are also useful in comparing trends of different attributes of a data set.

We can compare the methods using a real data set. Field 1 of Data Set 4 has the shape of a trapezoid about twice as long in the north-south direction as in the east-west direction (see Figure 3.4a below). The contents of the file *Set4.196sample.csv* are loaded into the data frame `data.Set4.1` using the code in Appendix B.4. There is a strong north-south trend in sand content. The data for plotting with the function `persp()` must be in the form of a matrix, so we make use of the *Row* and *Column* data used to identify the sample locations (Appendix B.4) to construct a matrix using a simple `for` loop.

```
> Sand <- matrix(nrow = 13, ncol = 7)
> for (i in 1:86){
+   Sand[data.Set4.1$Row[i], data.Set4.1$Column[i]] <-
+      data.Set4.1$Sand[i]
+ }
```

The percent sand values range up to about 45%. Rather than using the `scale` argument of the function `persp`, we simply scale the figure manually by multiplying the row and column numbers of the sample locations by 3, again obtained by trial and error.

```
> North <- 3 * 1:13
> West <- 3 * 1:7
> persp(North, West, Sand, theta = 30, phi = 20,
+   zlim = c(0,45), scale = FALSE) # Fig. 3.3a
```

The resulting perspective plot is shown in Figure 3.3a. Because of the curvature of the plot in the y (north-south) direction, it appears that a linear regression model fit should include quadratic terms, at least in this direction. We again use the function `lm()` to fit the trend.

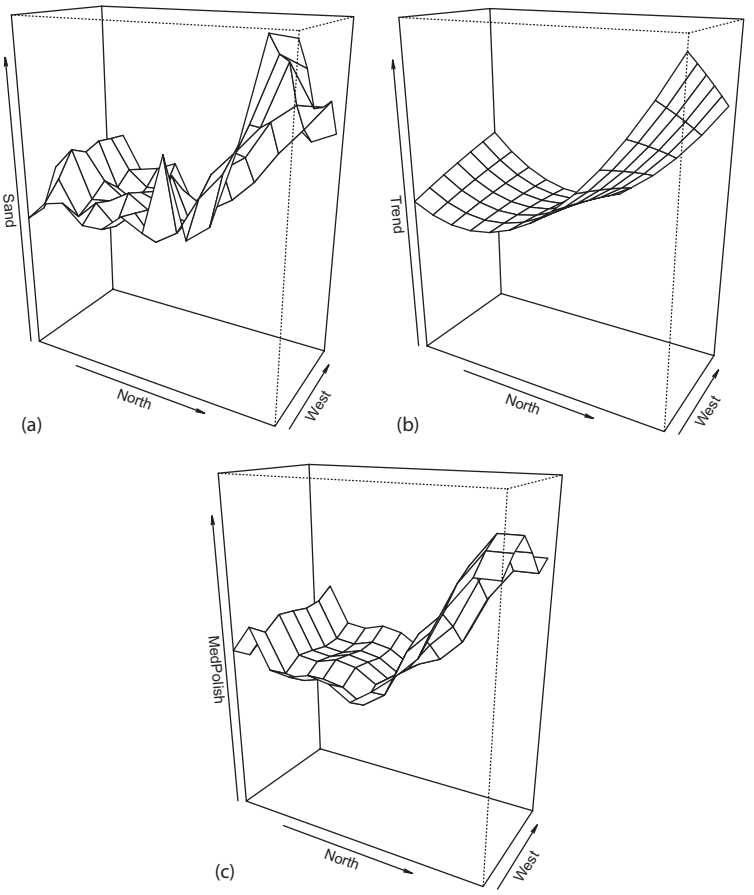

FIGURE 3.3
Perspective views of the sand data of Field 1 of Data Set 4: (a) actual data; (b) trend surface estimated by linear modeling including second-order terms; (c) trend surface estimated by median polish.

```
> trend.lm <- lm(Sand ~ Row + Column + I(Row^2) +
+   I(Column^2) + I(Row*Column), data = data.Set4.1)
```

The expressions such as `I(Row^2)` are used to indicate the evaluation of a function such as Row^2. The fitted model is $\hat{T}(x,y) = 22.42 + 1.29x - 1.95y - 0.21xy + 0.28y^2$; the x^2 coefficient is very small and has been omitted. This function captures the main feature of the trend: its rapid increase in the y direction at the south end of the field. The slight increase in the x (east-west) direction in the north end as well as the interaction may well be spurious, but in any case, they are not pronounced (Figure 3.3b). The median polish approximation (Figure 3.3c) appears to fit the data better than the linear model in the y direction and about the same in the x direction (where there is little if any trend).

One of the main reasons for computing the trend often is to remove it. Figure 3.4a shows a bubble plot of Field 4.1 in which the size of the circle is proportional to the sand content.

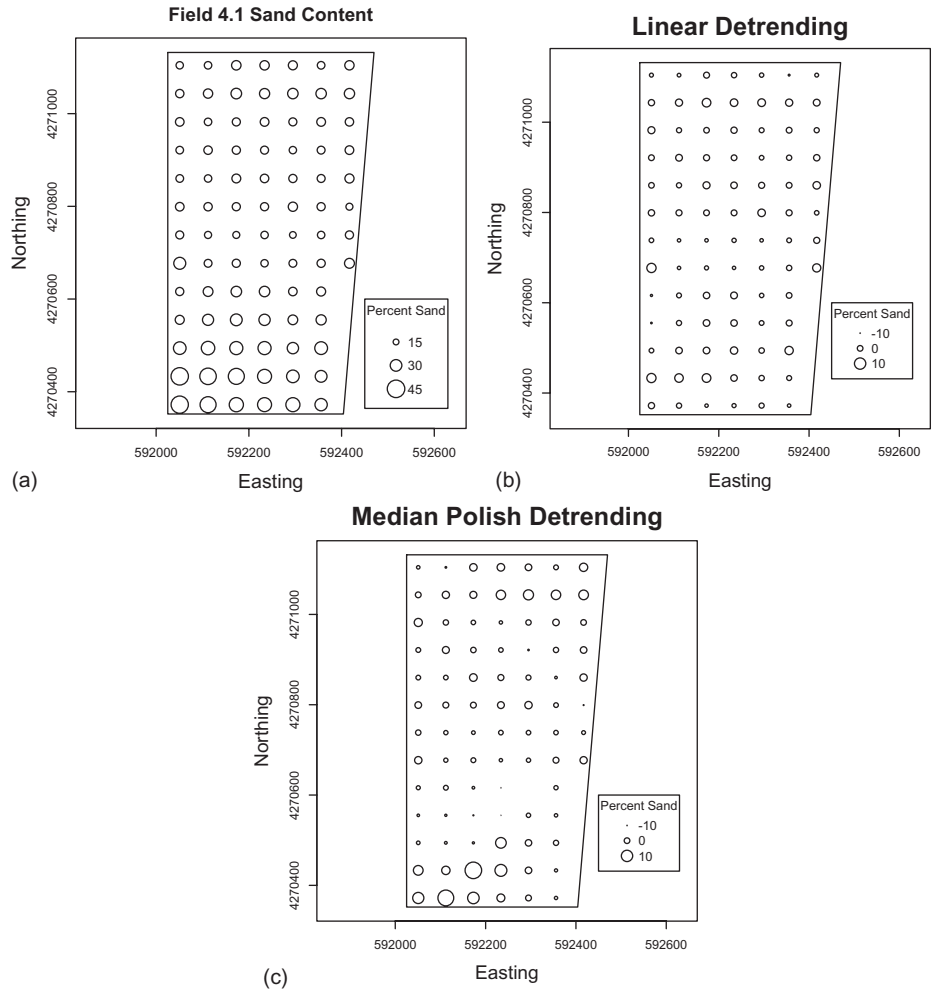

FIGURE 3.4
(a) Thematic bubble maps of sand content data of Field 1 of Data Set 4: (actual data); (b) data after trend removal using the linear model; (c) data after trend removal using median polish. The size of the circles represent the sand content values.

Equation 3.1 indicates that the quantity $\hat{\eta}(x,y)+\hat{\varepsilon}(x,y)=Y(x,y)-\hat{T}(x,y)$ (here as well as generally in this book the hats are used to indicate estimated values) should contain little if any trend; this quantity is referred to as the *detrended data*. Figure 3.4b and c show the detrended sand content data based on linear model and on median polish, respectively. A desirable property of these detrended data is to be *stationary*. This concept is discussed in the next section.

3.2.2 Stationarity

As stated in Section 3.1, the data described in this book are conceptualized as being a realization of a random field, that is, of a set of random numbers each of which is associated with a spatial location. To say that a data set is a *realization* of a random field means that nature has assigned, according to some well-defined law, values to the random variables that make up this data set. In theory, the assignment could be carried out again according to the same law to produce a different set of values that would be a different realization. Generally, one can only observe one realization of a spatial data set, and thus if one is to infer any of the properties of the law by which the values are assigned, one must use the measurements at different locations of that one realization, analogous to how one uses replications in a controlled experiment (Haining, 1990, p. 33). If these measurements at multiple locations are to be considered as if they were replications, then the law by which they are assigned must not vary from one location to another. A spatial random process whose properties do not vary by location is said to be *stationary*.

We will use a coin-tossing simulation to gain an intuitive idea of stationarity. As with spatial stationarity described as location invariance in the previous paragraph, a time series is stationary if its statistical properties are invariant in time. We begin by simulating a sequence of 30 tosses, each of which has a probability 0.5 of landing heads. Since the probability is independent of time, the process is stationary. Instead of simulating the entire set of coin tosses as a single random vector, we will use a `for` loop to explicitly simulate each toss. We use the function `rbinom()`, which is included in the base R package (try `?rbinom`).

```
> set.seed(123)
> n.tosses <- 30
> head <- numeric(n.tosses)
> for (i in 1:n.tosses) head[i] <- rbinom(1,1,0.5)
> head
 [1] 0 1 0 1 1 0 1 1 1 0 1 0 1 1 0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0
```

In the sequence of tosses, every value has an equal probability of turning up heads (one) or tails (zero). By chance there is a very long run of heads towards the end of the sequence. Now let us make each toss dependent on the previous toss.

```
> set.seed(123)
> n.tosses <- 30
> head <- numeric(n.tosses)
> p <- 0.5
> for (i in 1:n.tosses){
+   head[i] <- rbinom(1,1,p)
+    p <- ifelse(head[i] > 0, 0.8, 0.2)
```

```
+ }
> head
  [1] 0 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0
```

This sequence is also stationary. It is true that, for example, in this data set the probability that the fifth toss will be a head is 0.8, because the fourth toss was a head, while the probability that the sixth toss will be a head is 0.2, because the fifth toss was a tail. However, the same rule for determining the outcome of the toss, a probability of 0.8 that it will be the same as the previous toss, applies for all tosses, and this independence of time of the toss is what determines stationarity in a time series.

Next, suppose that the probability that a toss lands heads starts at a low value gradually increases as the sequence proceeds.

```
> set.seed(123)
> n.tosses <- 30
> head <- numeric(n.tosses)
> p <- 0.3
> for (i in 1:n.tosses){
+   head[i] <- rbinom(1,1,p)
+   p <- p + 0.02
+ }
> head
  [1] 0 1 0 1 1 0 0 1 1 0 0 1 0 0 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1
```

This sequence is not stationary, because the probability of heads depends on position in the sequence.

It would be very hard to determine based on the data which of the sequences in these three examples is stationary and which is not. Moreover, if in the second example the probabilities of head and tail were 0.99 and 0.01 instead of 0.8 and 0.2, it could well be the case that the data would consist of a sequence of, for example, only heads followed by a sequence of only tails. Nevertheless, this would be a stationary process. We can recall in this context the comment of Cressie (1991) quoted in Section 3.2.1 that one person's deterministic trend is another person's correlated error structure. We will have to confront this issue again and again over the course of our analyses.

Now we return to spatial processes. For the present, we define stationarity only for areal data such as that represented in Figure 1.3b. The most useful definition of stationarity for areal data is *second-order stationarity*. A random field Y defined on a finite mosaic consisting of n locations is second order stationary if (Cliff and Ord, 1981, pp. 142–143, see also Anselin, 1988, p. 42, Isaaks and Srivastava, 1989, p. 222)

$$\begin{aligned} E\{Y_i\} &= \mu, \\ \text{var}\{Y_i\} &= \sigma^2, \\ \text{cov}\{Y_i, Y_j\} &= \sigma^2 cor\{(x_i, y_i) - (x_j, y_j)\}, \end{aligned} \tag{3.4}$$

where μ and σ^2 are independent of location i, (x_i, y_i) and (x_j, y_j) are the position vectors of locations i and j, and $cor(x, y)$ is a correlation function depending only on the relative locations of (x_i, y_i) and (x_j, y_j), and not on their specific positions. A stronger definition of stationarity, called *strict stationarity*, requires that the independence of location expressed

in Equations 3.4 hold for all moments of the distribution. If the distribution is normal, then second-order stationarity implies strict stationarity since the normal distribution is defined by its first two moments, the mean and the variance, but this is not true for probability distributions in general. However, second-order stationarity ensures that two of the most important statistical properties, the mean and the variance, do not vary with position. Our original motivation for requiring stationarity was to use multiple measurements as replications. In this context, second-order stationarity is sufficiently strong to serve the purpose, but not so strong as to exclude data sets that we want to be able to consider stationary.

Sometimes there are obvious reasons why spatial data are not stationary. Perhaps the most common example is if the data involve species dispersal. By definition, the interesting part of these data is generally how their pattern changes in time, and a population that is stationary is not dispersing. Another example is if there is an obvious gradient in some important quantity. For example, the oak distribution data in Data Set 1 are unlikely to be stationary since the trees are heavily influenced, either directly or indirectly, by elevation, which is on a spatial gradient. The exploratory methods described in Chapters 7 through 9 do not require spatial stationarity since they are non-spatial in nature, but the spatially based methods described in Chapters 13 and after do assume stationarity in some sense. In the case of regression methods, however, often it is not stationarity of the data that is required but rather stationarity of the residuals. Sometimes the residuals of a properly parameterized regression model will be stationary (or at least close enough) even if the data are not. Sometimes they are not, however, and so some means of testing for stationary can be very important.

There is an extensive literature on testing for stationarity in time series (e.g., Elliot et al., 1996). Ultimately, however, stationarity is a modeling assumption and, since there are many ways in which a process can be non-stationary, there is no test that can cover all of them. There has been less comparable development of tests of stationarity for spatial data than for time series. There are two fundamental differences between time series data and spatial data that make the notion of testing for stationarity of spatial data less meaningful (Haining, 2003, p. 47). The first is that time is characterized by a flow (and hence a dependency) in one direction. One can distinguish between past, present, and future: one can specify that events in the present or past cannot depend on the future. There is no equivalent concept with spatial data. Second, spatial data $Y(x, y)$ exist (in this book) in two dimensions, and there is no guarantee that the structure of spatial data be the same in one dimension as it is in the other. If the dependency has the same structure independent of direction, the data are said to be *isotropic*, otherwise the data are *anisotropic*. If you rub your hand along a piece of corduroy fabric, the resistance depends on whether you rub with the cords or perpendicular to them, so it is anisotropic. Plain fabric does not offer a different resistance depending on direction; it is isotropic.

The properties of stationarity and isotropy in spatial data must always be presented as assumptions, which can be examined in an exploratory way but cannot be subjected to rigorous hypothesis testing. As stated succinctly by ver Hoef and Cressie (2001, p. 299), "these assumptions are impossible to test, because it is impossible to go back in time again and again and generate the experiment each time to check whether each experimental unit has the same mean value or whether the correlation is the same for all pairs of plots that are at some fixed distance from each other. However, any gross spatial trends in the residuals…would cause suspicion that they are not stationary." A data set like the sand content data discussed in Section 3.2.1 obviously is not stationary, although the residuals of the detrended data (i.e., the sum $\hat{\eta}(x, y) + \hat{\varepsilon}(x, y)$) might be

expected to satisfy the assumption of stationarity. There are some statistical tools that can be used to explore the stationarity of spatial data. One of these is the local Moran's I, discussed in Section 4.5.3, and another is geographically weighted regression, discussed in Section 4.5.4.

3.3 Monte Carlo Simulation

In many cases, the properties of a statistical model cannot be worked out analytically. *Monte Carlo simulation* is a very useful tool for estimating these properties numerically. Monte Carlo simulation has a number of definitions (Ripley, 1981, Besag and Clifford, 1989, Manly, 1997), but a very general one is given by Rubinstein (1981, p. 11) as "a technique, using random or pseudorandom numbers, for solution of a model." The statistical definition of an experiment is an action that can in principle be replicated an arbitrary number of times (Larsen and Marx, 1986, p. 14). We have made a point of distinguishing between a replicated experiment and an observational study, so at the risk of being pedantic we will put the word "experiment" in quotes when used in the statistical context just given. In practice "experiments" are generally only carried out once or a few times. The idea of a Monte Carlo simulation is to carry out a simulated "experiment" many (e.g., 1000 or 10,000) times, to observe the properties of the resulting distribution of outcomes, and to compute statistics characterizing this distribution. It is usually no problem for R to generate the pseudorandom numbers (see Section 2.3.2) needed to simulate the "experiment" many times.

The R function `replicate()` can be used to carry out Monte Carlo simulations. Consider the "experiment" consisting of tossing a fair coin $n = 20$ times and recording the number of heads. The probability of the coin turning up heads on any one toss is $p = 0.5$. According to standard statistical theory (Larsen and Marx, 1986, p. 96), the number of heads in 20 tosses follows a binomial distribution with a mean $np = 10$ and a variance $np(1-p) = 5$. Let us implement and display the results of a Monte Carlo simulation of this "experiment." The actual "experiment" is placed in a function called `coin.toss()` that generates the number of heads as a pseudorandom number from a binomial distribution.

```
> coin.toss <- function (n.tosses, p){
+   n.heads <- rbinom(1,n.tosses, p)
+ }
```

The function takes two arguments, the number of tosses `n.tosses` and the probability of heads `p`. Note the lack of a `return` statement. As mentioned in Section 2.5, when there is no `return` statement, the last quantity computed, which in this case is `n.heads`, is returned.

Now we are ready to carry out the Monte Carlo simulation.

```
> set.seed(123)
> n.tosses <- 20
> p <- 0.5
> n.reps <- 10000
> U <- replicate(n.reps, coin.toss(n.tosses, p))
```

The function `replicate()` generates a vector `U` whose elements are the number of heads in each of the 10,000 replications of the experiment.

```
> mean(U)
[1] 9.9814
> var(U)
[1] 4.905345
> hist(U, cex.lab = 1.5, # Fig. 3.5
+     main = "Number of Heads in 20 Tosses",
+     cex.main = 2, xlab = "Number of Heads")
```

The mean and variance of `U` are close to their theoretically predicted values of 10 and 5, respectively, and a histogram of `U` indicates that it has the approximately normal distribution (Figure 3.5) that is expected based on central limit theorem (Larsen and Marx, 1986, pp. 206, 322). This theorem says, roughly speaking, that, given a set of random variables from any distribution, not necessarily normal, the sample mean of this set is approximately normally distributed, with the approximation becoming better as the sample size increases.

In the simple example of the coin tossing "experiment" just carried out, the statistical properties of the sampling distribution (i.e., its mean, variance, etc.) could be computed analytically, but for many "experiments," particularly those involving spatial statistics, this is not the case. In such circumstances, Monte Carlo simulation may be the only possibility for obtaining insight into the properties of the data. Even in those cases where a closed form solution is possible, a Monte Carlo simulation often provides insight and intuition not so easily available from analytical calculations.

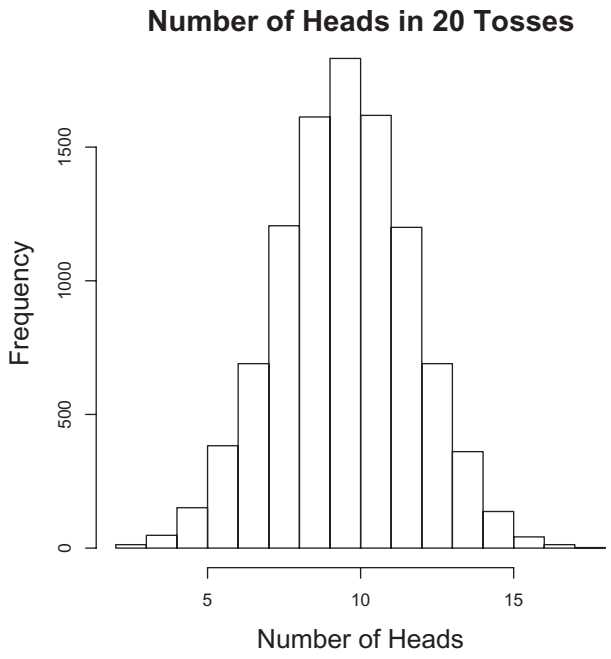

FIGURE 3.5
Histogram of the results of 10,000 Monte Carlo simulations of the tossing of a fair coin 20 times in succession and counting the number of heads.

3.4 A Review of Hypothesis and Significance Testing

Spatial autocorrelation often affects the outcome of significance tests. For this reason, we will spend some time reviewing this procedure. The review is informal and intuitive; for a more formal discussion see Muller and Fetterman (2002, p. 4) and Schabenberger and Pierce (2001 p. 22). Suppose we have measured a set of values $\{Y_1, Y_2, ..., Y_n\}$ of a normally distributed random variable Y whose mean μ and variance σ^2 are unknown. In this book, both the random variable and its values are denoted with an upper case Latin letter. Other texts frequently denote the values of the random variable with lowercase letters, but we do not follow this practice because the symbols x and y are used to indicate spatial coordinates. We can write the expression for the Y_i as

$$Y_i = \mu + \varepsilon_i,\ i = 1, ..., n, \tag{3.5}$$

where the ε_i are independent, identically distributed random variables drawn from a normal distribution with mean zero and variance σ^2. We wish to test the null hypothesis that the value of the mean μ is zero against the alternative that it is not,

$$\begin{aligned} H_0 &: \mu = 0, \\ H_a &: \mu \neq 0. \end{aligned} \tag{3.6}$$

When the population from which the values Y_i are drawn is normal, this hypothesis can be tested using the Student t statistic. This statistic is defined as follows. The sample mean $\bar{Y}$ is given by

$$\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i. \tag{3.7}$$

If the Y_i are independent, then the variance of $\bar{Y}$ is given by

$$\text{var}\{\bar{Y}\} = \frac{\sigma^2}{n} \tag{3.8}$$

(Larsen and Marx, 1986, p. 321). Moreover, the sample variance, defined by

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Y_i - \bar{Y})^2, \tag{3.9}$$

is an unbiased estimator of σ^2. Therefore s^2 / n is an unbiased estimator of the variance of $\bar{Y}$. The square root of this quantity is called the *standard error* and is given by

$$s\{\bar{Y}\} = \frac{s}{\sqrt{n}}. \tag{3.10}$$

With these quantities defined, the Student t statistic is given by

$$t = \frac{\bar{Y} - \mu}{s\{\bar{Y}\}}. \tag{3.11}$$

TABLE 3.1

The Four Possible Outcomes of the Test $H_0 : \mu = 0$ against the Alternative $H_a : \mu \neq 0$

	H_0 True	H_0 False
Don't reject H_0	OK	Type II error
Reject H_0	Type I error	OK

The distribution of this statistic under $H_0 : \mu = 0$ in (3.5) is called the Student t distribution.

One way to carry out the test is to compare the value of the t statistic computed in Equation 3.11 with the percentiles of the Student t distribution. If H_0 is true, then the value of t should be small. According to the standard theory (Larsen and Marx, 1986, p. 343), if we define $t_{\alpha/2}$ as the $\alpha/2$ percentile of the Student t distribution, then $P\{\mid t \mid > t_{\alpha/2}\} = \alpha$, i.e., the probability that the magnitude of the t statistic will exceed the $\alpha/2$ percentile, is equal to α. There are four possible outcomes of this test, which are shown in Table 3.1. Two of these represent an error, that is, a failure to draw the correct conclusion from the test. In particular, the test will with probability α result in a Type I error, that is, a rejection of the null hypothesis when it is true (Table 3.1). It is important to emphasize that the most appropriate interpretation of a failure to reject the null hypothesis is *not* to conclude that the null hypothesis is true and that $\mu = 0$. We have not proven that $\mu = 0$; we have simply not been able to demonstrate with adequate certainty that it is not zero. The most appropriate way to verbalize a failure to reject the null hypothesis that $\mu = 0$ is to say that, based on our available evidence, we cannot distinguish the value of μ from zero.

There are two approaches to the test of H_0 (Schabenberger and Pierce, 2001, p. 22). The first is to fix α at some pre-assigned value, say, 0.1 or 0.05, and determine based on the value of the percentile of the t distribution relative to this fixed value whether or not to reject H_0. The second approach is to report the probability of observing a statistic at least as large in magnitude as that actually observed (i.e., the p value) as the *significance* of the test. Manly (1997, p. 1) advocates the latter policy, pointing out that "To avoid the characterization of belonging to 'that group of people whose aim in life is to be wrong 5% of the time' (Kempthorne and Doerfler, 1969), it is better to regard the level of significance as a measure of the strength of evidence against the null hypothesis rather than showing whether the data are significant or not at a certain level." Nevertheless, the magic number 0.05 is sufficiently ingrained in applied statistics that it is impossible to completely avoid using it. In addition, we will see that it provides a convenient means to measure the effect of spatial autocorrelation on the results of the test.

If, for whatever reason, the null hypothesis H_0 is not rejected, there is the possibility that this decision is incorrect. Failure to reject the null hypothesis when it is false is called a Type II error (Table 3.1). The *power* of the test is defined as the probability of correctly rejecting H_0 when it is false, that is, $power = 1 - \Pr\{Type\ II\ error\}$.

As an illustration of the use of R in a test of a null hypothesis such as that in Equation 3.6, we generate a sample of sample size $n = 10$ from a standard normal distribution.

```
> set.seed(123)
> Y <- rnorm(10)
> Y.ttest <- t.test(Y, alternative = "two.sided")
> Y.ttest$p.value
 [1] 0.8101338
```

It happens that this time the p value is about 0.81. We can modify this to carry out a Monte Carlo simulation of the test. The default value α of the function `ttest()` is 0.05, and indeed the fraction of Type I errors is close to this value.

```
> set.seed(123)
> ttest <- function(){
+   Y <- rnorm(10)
+   t.ttest <- t.test(Y,alternative = "two.sided")
+   TypeI <- t.ttest$p.value < 0.05
+ }
> U <- replicate(10000, ttest())
> mean(U)
[1] 0.0485
```

There is an alternative method for carrying out hypothesis and significance tests that we shall occasionally employ. This is called, depending on the author, either a *permutation test* or a *randomization test*. We will use the former term, although by doing so we probably put ourselves in the minority, in order to avoid confusion with the *randomization assumption*, which is discussed in Chapter 4. Permutation tests are nonparametric, so that they do not depend on the population fitting any parameterized distribution, in the way that the t test depends on the normal distribution (see Exercise 3.1). They are simple both to understand and to carry out. For a reason that will soon become apparent, we will not use a permutation test to test the null hypothesis of Equation 3.6. Instead, we introduce this form of hypothesis test using a *two-sample* test, in which we test the hypothesis that two independent samples come from the same probability distribution (Manly, 1997, p. 97). We first draw two random sequences $Y_1 = \{Y_{1i}\}$, $i = 1,...,5$ and $Y_2 = \{Y_{2i}\}$, $i = 1,...,5$ of five values each. Each sequence is drawn from a standard normal distribution (since this test is nonparametric, the parameters could be drawn from any distribution, but the normal is a convenient one). The difference d between the means of the two sequences is small but not negligible.

```
> set.seed(123)
> print(Y1 <- rnorm(5))
[1] -0.56047565 -0.23017749 1.55870831 0.07050839 0.12928774
> print(Y2 <- rnorm(5))
[1] 1.7150650 0.4609162 -1.2650612 -0.6868529 -0.4456620
> print(d <- mean(Y1) - mean(Y2))
[1] 0.2378892
```

We will test the hypothesis that the expected values of the parent distributions of the two samples are the same. The basic idea is to repeatedly create new vectors Y_1' and Y_2' by rearranging (i.e., permuting) the elements of Y_1 and Y_2, and comparing the mean of the differences $d' = Y_1' - Y_2'$ with that of $d = Y_1 - Y_2$. If Y_1 and Y_2 come from the same population then, if we repeat this permutation many times, the difference d between the means of Y_1 and Y_2 should not have an extreme value in the distribution of means of permuted arrays. If it does have an extreme value, then it is likely that Y_1 and Y_2 do not come from the same population.

The first step of the procedure is to concatenate the two sequences into a single sequence of 10 elements. Then we rearrange these in a random order value and finally we separate the reordered array into two new sub-arrays of five elements each. These are the test arrays Y_1' and Y_2'. We use the R function `sample()` to implement the permutation. Read about this

function using `?sample`. By default `sample(x,...)` returns an array of the same length as `x` of randomly sampled values of the elements of `x`. One of the optional arguments of `sample()` is `replace`, which indicates sampling with replacement if it has the value `TRUE`. By default the value of `replace` is `FALSE`.

```
> Y <- c(Y1, Y2)
> Ysamp <- sample(Y,length(Y))
> print(Yprime1 <- Ysamp[1:5])
[1] -0.68685285 0.46091621 1.71506499 -0.44566197 0.07050839
> print(Yprime2 <- Ysamp[6:10])
[1] -1.2650612 1.5587083 -0.2301775 -0.5604756 0.1292877
> print(dprime <- mean(Yprime1) - mean(Yprime2))
[1] 0.2963386
```

The difference $d' = Y_1' - Y_2'$ in this case happens to be about the same magnitude as d. Now we create a function `perm.diff()` that carries out the permutation and computation of d'

```
> perm.diff <- function(Y1, Y2){
+   Y <- c(Y1, Y2)
+   Ysamp <- sample(Y,length(Y))
+   Yprime1 <- Ysamp[1:5]
+   Yprime2 <- Ysamp[6:10]
+   dprime <- mean(Yprime1) - mean(Yprime2)
+ }
```

We can now use the function `replicate()` to generate a set of differences between the means of permutations of the elements of Y_1 and Y_2. We start with nine permutations and add the observed difference d as the tenth.

```
> set.seed(123)
> U <- replicate(9, perm.diff(Y1, Y2))
> sort(c(d,U))
 [1] -0.64644597 -0.61387764 -0.20122761 -0.08870639 -0.08856700
 [6]  0.15530214  0.23788923  0.29633862  0.86680338  1.14863761
```

The observed d occupies the seventh position among the differences in permutations. To get a more precise evaluation, we can generate a p value by running a large number (say, 10,000) permutations and counting the number in the "tail," that is, the number of values that are farther away from the median than is d. Doubling this number (to account for both tails) and dividing by 10,000 yields a p value.

```
> set.seed(123)
> U <- replicate(9999, perm.diff(Y1, Y2))
> U <- c(U, d) # Add original observation to get 10,000
> U.low <- U[U < d] # Obtain diff. values less than d
> {if (d < median(U)) # Is d in the upper or lower tail?
+    n.tail <- length(U.low)
+  else
+    n.tail <- 10000 - length(U.low)
+ }
> print(p <- 2 * n.tail / 10000) # Two tail test
[1] 0.714
```

The elements of the vector `U` are the 9,999 differences between means of random permutations of the values in Y_1 and Y_2; the original value `d` is the ten thousandth. The statement `U.low <- U[U < d]` is used to determine how many values of `U` are less than the difference between the original samples. If this value is either very small or very large, then d is judged sufficiently extreme to justify rejection of the null hypothesis. In the example, about 71% of the randomized rearrangements have differences farther away from the middle value than the observed value of 0.238 (to three decimal places) so the p value computed by the permutation test is $p = 0.71$. This is not far from the value computed by a t test.

```
> t.test(Y1, Y2, "two.sided")$p.value
[1] 0.7185004
```

Having carried out the two sample permutation test, we can see why a one sample permutation test would not work. Rearranging the order of a single sample does not affect the sample mean, so there is nothing to test (Manly, 1997, p. 17). There are other nonparametric tests that can be used for a one sample test, the simplest of which is the *sign test* (Crawley, 2007, p. 300), but we will not pursue these here.

There has been considerable discussion over the years about the scope of inference of permutation tests and how it compares with the scope of inference of traditional parametric tests such as the t-test. Our summary of this discussion is based on comments by Romesburg (1985) and Edgington (2007). One of the assumptions underlying parametric tests such as the Student t-test is that the data are a random sample of the population, and in many cases this assumption is violated. Permutation tests, on the other hand, do not require this randomness assumption. Fisher (1935) originally developed the permutation test as a means of demonstrating the robustness of the Student t-test when used with non-normal data. Romesburg (1985, p. 22) asserts that because of the close correspondence between the results of parametric tests and permutation tests, a parametric test may be considered as an approximation of a permutation test. In this context, the assumption of randomness can, according to Romesburg (1985), be relaxed for the parametric test as well. On the other hand, Edgington (2007) and others have pointed out that, strictly speaking, permutation tests apply only to the data sets on which they are based, and cannot with statistical validity be extended to a wider population. In other words, the results of an analysis using parametric statistics can be extended to the entire population from which the sample is drawn, but require assumptions of random sampling that are almost never satisfied in practice. The results of a permutation test, on the other hand, do not depend on assumptions of random sampling but cannot be extended to the entire population from which the sample is drawn. What does one do in the face of this dilemma? First, when both parametric and permutation tests can be applied, the best approach is to use them both and compare the results. When one does not have a theoretical justification for extending one's scope of inference, one must be as careful as possible to do everything practical to justify this extension in scope, that is, to make sure that the data set is not selected from the population in a way that makes it a biased sample either because the selection process was biased or because the scope of inference is extended too far. It is useful to recall a quote of G.E.P. Box (1976): "Since all models are wrong, the scientist must be alert to what is importantly wrong. It is inappropriate to be concerned about mice when there are tigers abroad." It is, however, entirely appropriate to do everything possible to ensure that no tigers are inadvertently released.

3.5 Modeling Spatial Autocorrelation

3.5.1 Monte Carlo Simulation of Time Series

We begin the discussion of the simulation of spatial data with a simpler topic: the simulation of time series data. We will use time series to illustrate the effects of positive autocorrelation on the outcome of hypothesis tests. Much of the theory of spatial statistics grew out of the analogy with time series (Whittle, 1954; Bartlett, 1935, p. 18) and because of the one-dimensional and directional nature of time series data, many of the most important concepts are more intuitive in that domain. Simulation through the use of artificially generated data sets has the advantage of permitting the analysis of data whose distributional properties are known. We will use Monte Carlo simulation to explore the effect of temporal autocorrelation on the outcome of a test of the null hypothesis that an observed data set is drawn from of a population with mean zero.

In Section 3.4, we presented the results of a simulation using uncorrelated data. The simulated error rate, that is, the fraction of times a Type I error occurred, was 0.0485.

Now we will explore the effect of autocorrelation on the outcome of the *t*-test. The autocorrelation takes the form of a first-order autoregressive time series in the error terms (Kendall and Ord, 1990, p. 56). The model is initially written as

$$\begin{aligned} Y_i &= \mu + \eta_i \\ \eta_i &= \lambda \eta_{i-1} + \varepsilon_i,\ i = 1, 2, \ldots \end{aligned} \tag{3.12}$$

where $-1 < \lambda < 1$ and the ε_i are independent normally distributed random variables with mean zero and variance σ^2. The η_i are the autoregressive error terms. The model is specified in this form to maintain consistency with the spatial model that will be discussed in the next section.

The equations are easiest to work with if we substitute $\eta_i = Y_i - \mu$ and $\eta_{i-1} = Y_{i-1} - \mu$ into the second of Equations 3.12 to get

$$Y_i - \mu = \lambda(Y_{i-1} - \mu) + \varepsilon_i,\ i = 1, 2, \ldots \tag{3.13}$$

If $\lambda = 0$ then this reduces to Equation 3.5. The time series is initiated by specifying Y_1 and setting $\eta_0 = 0$ and $\varepsilon_1 = 0$. We then have

$$\begin{aligned} Y_2 &= \mu + \lambda(Y_1 - \mu) + \varepsilon_2 \\ Y_3 &= \mu + \lambda(Y_2 - \mu) + \varepsilon_3 \\ &= \mu + \lambda^2(Y_1 - \mu) + \lambda\varepsilon_2 + \varepsilon_2 \\ Y_4 &= \mu + \lambda(Y_3 - \mu) + \varepsilon_4 \\ &= \mu + \lambda^3(Y_1 - \mu) + \lambda^2\varepsilon_2 + \lambda\varepsilon_3 + \varepsilon_4. \end{aligned} \tag{3.14}$$

and so forth. Since $-1 < \lambda < 1$, $\lambda^j \to 0$ as j increases, so that the influence of the initial term declines with each time step.

We will simulate this process with the value of μ set to zero so that the null hypothesis is true. Since $\mu = 0$, it follows that Equation 3.13 becomes

$$Y_i = \lambda Y_{i-1} + \varepsilon_i,\ i = 1, 2, \ldots, \tag{3.15}$$

and this is how the simulation is programmed. The first simulation, with λ set at 0.4, involves a sample of size 10. The initial value is set at $Y_1 = 0$, which is a fixed, non-random number. This may cause the data to be non-stationary for the first few values of i. This effect is sometimes called an "initial transient," and to avoid it, twenty values of Y_i are generated and the first ten are discarded.

```
> lambda <- 0.4 #Autocorrelation term
> set.seed(123)
> Y <- numeric(20)
> for (i in 2:20) Y[i] <- lambda * Y[i - 1] + rnorm(1)
> Y <- Y[11:20]
```

There is an R function `arima.sim()` that accomplishes the same thing as this code without using a for loop. The call to the function in this case would be `Y <- arima.sim(list(ar=lambda),n = 10, n.start = 10)`. The for loop is used explicitly to emphasize the connection with Equation 3.15.

We can now carry out a *t*-test of the null hypothesis $H_0 : \mu = 0$.

```
> Y.ttest <- t.test(Y,alternative = "two.sided")
> #Assign the value 1 to a Type I error
> TypeI <- as.numeric(Y.ttest$p.value < 0.05)
> Ybar <- mean(Y)
> Yse <- sqrt(var(Y) / 10)
> c(TypeI, Ybar, Yse)
[1] 0.0000000 0.2790157 0.3028951
```

We concatenate into a single vector an indicator variable for a Type I error (1 = Type I error, 0 = H_0 not rejected), the sample mean $\bar{Y}$ and standard error $s\{\bar{Y}\}$.

Now we are ready to combine this into a Monte Carlo simulation.

```
> set.seed(123)
> lambda <- 0.4
> ttest <- function(lambda){
+   Y <- numeric(20)
+   for (i in 2:20) Y[i] <- lambda * Y[i - 1] + rnorm(1, sd = 1)
+   Y <- Y[11:20]
+   Y.ttest <- t.test(Y,alternative = "two.sided")
+   TypeI <- as.numeric(Y.ttest$p.value < 0.05)
+   Ybar <- mean(Y)
+   Yse <- sqrt(var(Y) / 10)
+   return(c(TypeI, Ybar, Yse))
+ }
```

We use the function `replicate()`to carry out the simulation as described in Section 2.6.

```
> U <- replicate(10000, ttest(lambda))
> mean(U[1,]) # Type I error rate
[1] 0.1936
> mean(U[2,]) # Mean value of Ybar
[1] 0.003212074
> mean(U[3,]) # Mean est. standard error
[1] 0.3128384
> sd(U[2,]) # Sample std. dev. of Ybar
[1] 0.5028021
```

The return value of the function `ttest()` is `c(TypeI, Ybar, Yse)`. This is interpreted by R as a column vector, and thus the output of `replicate()` is a $3\times10,000$ matrix (i.e., each column of the matrix is one replication).

The fraction of Type I errors in the 10,000 simulation runs is 0.194, well above the nominal error rate of 0.05. The effect of the positive autocorrelation has been to make the *t*-test dramatically more "liberal" (sometimes called "anti-conservative"), that is, to reject the null hypothesis more often than the theory predicts. Figure 3.6a shows a plot of the Type I error rate of the hypothesis test of Equation 3.6 applied to the model of Equation 3.15 as a function of λ. The error rate increases dramatically with increasing λ.

Why does this happen? The outcome of the test is based on the value of the t statistic in Equation 3.11. From this equation, we can see that since the null hypothesis specifies $\mu = 0$ there are two possible explanations for an increased Type I error rate: either the magnitude of the numerator $\bar{Y}$ is overestimated (i.e., the estimator $\bar{Y}$ is biased), or the denominator $s\{\bar{Y}\}$, the standard error, underestimates the true standard deviation of $\bar{Y}$ (or both). Based on Figure 3.6b, the numerator $\bar{Y}$ appears to be estimated as approximately zero. However, comparing Figure 3.6c and 3.6d indicates that although the value of $s\{\bar{Y}\}$ as computed in Equation 3.10 stays approximately constant and near its theoretical value of $1/\sqrt{10} = 0.316$, the actual standard deviation of $\bar{Y}$ increases dramatically as λ increases from zero.

It is not hard to show that the estimate $\bar{Y}$ of the population mean is unbiased whether or not the errors are autocorrelated. Indeed, from Equation 3.7, since ε_i has mean 0 and variance σ^2, we have

$$\begin{aligned} E\{\bar{Y}\} &= E\left\{\sum_{i=1}^{n}\frac{Y_i}{n}\right\} = E\left\{\frac{1}{n}\left[\sum_{i=1}^{n}\mu + \lambda(Y_{i-1}-\mu)+\varepsilon_i\right]\right\} \\ &= \frac{n\mu}{n} + E\left\{\sum_{i=1}^{n}\lambda(Y_{i-1}-\mu)\right\} + \frac{1}{n}\sum_{i=1}^{n}E\{\varepsilon_i\} \\ &= \mu + \lambda\sum_{i=1}^{n}E\{Y_{i-1}-\mu\} + 0 \\ &= \mu + \lambda\times 0 + 0 \\ &= \mu \end{aligned} \tag{3.16}$$

The variance of the mean, on the other hand, is (Haining, 1988)

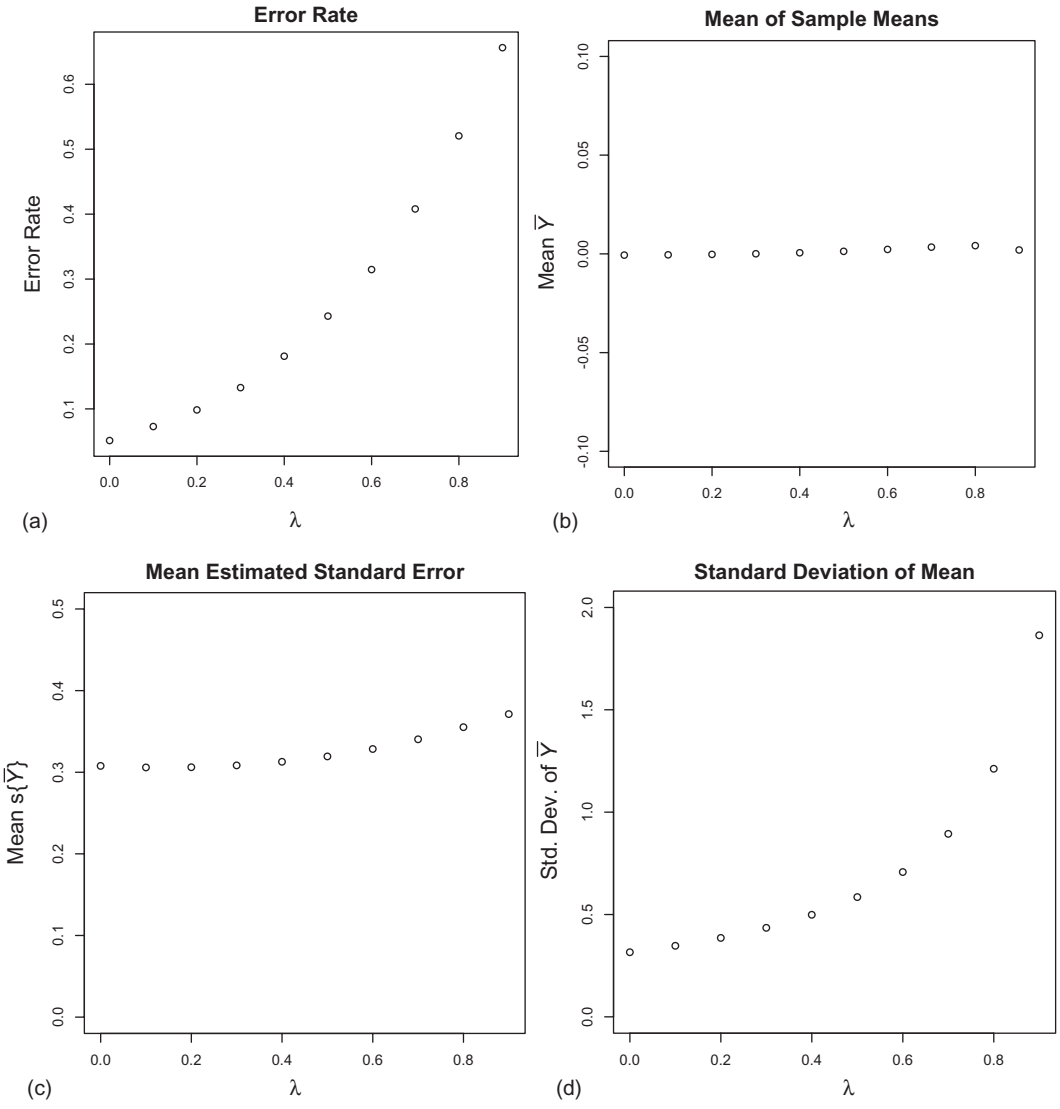

FIGURE 3.6
Results of Monte Carlo simulations of an autoregressive time series model with parameter λ for $0 \le \lambda \le 0.8$. Plotted against values of λ are: (a) Type I error rate (fraction of times a Type I error is made) for of a test of the null hypothesis $H_0 : \mu = 0$ against the alternative $H_a : \mu \neq 0$; (b) mean value of $\bar{Y}$ over the 10,000 simulations; (c) mean value of $s\{\bar{Y}\}$ over the 10,000 simulations; (d) standard deviation of $\bar{Y}$ over the 10,000 simulations.

$$\begin{aligned}\text{var}\{\bar{Y}\} &= \text{var}\left\{\frac{1}{n}\sum_i Y_i\right\} \\ &= \frac{1}{n^2}\sum_i \text{var}\{Y_i\} + \frac{2}{n^2}\sum_i \text{cov}\{Y_i, Y_{i-1}\} \\ &= \frac{\sigma^2}{n} + \frac{2}{n^2}\sum_i \text{cov}\{Y_i, Y_{i-1}\}.\end{aligned} \tag{3.17}$$

Therefore, if $\text{cov}\{Y_i, Y_{i-1}\} > 0$ then $\text{var}\{\bar{Y}\} > \sigma^2/n$ and therefore the quantity σ^2/n in Equation 3.8 underestimates $\text{var}\{\bar{Y}\}$. Moreover, the expected value of s^2 defined in Equation 3.9 is

$$E\{s^2\} = \sigma^2 - \frac{2}{n(n-1)}\sum_i \text{cov}\{Y_i, Y_{i-1}\}. \tag{3.18}$$

(Haining, 1988). Therefore, if $\text{cov}\{Y_i, Y_{i-1}\} > 0$ then the quantity $s\{\bar{Y}\}$ of Equation 3.10 underestimates the standard deviation of $\bar{Y}$ through a combination of two effects on this standard deviation: first, because $s\{\bar{Y}\}$ in Equation 3.10 underestimates the square root of σ^2/n, and second, because σ^2/n, in Equation 3.8 underestimates $\text{var}\{\bar{Y}\}$ (Haining, 1988).

This effect of temporal autocorrelation, that the quantity $s\{\bar{Y}\} = s/\sqrt{n}$ underestimates the true standard deviation of $\bar{Y}$, means that the denominator of the t statistic in Equation 3.11 is smaller than it should be to properly take into account the variability of $\bar{Y}$, and therefore the t statistic is larger than it would be if the random variables Y_i were uncorrelated. The effect of inflating the value of the t statistic is to increase the Type I error rate, the fraction of times that the test exceeds the threshold for rejection of the null hypothesis. If, for example, the size of α is fixed at $\alpha = 0.05$ then the actual fraction of tests carried out in which the null hypothesis is rejected will actually be higher than 0.05, as demonstrated above.

Another way of interpreting this effect is that when data are autocorrelated, each data value provides some information about the other data values near it. When the data are independent, each of the n data values carries only information about itself and, in statistical terms, brings a full degree of freedom to the statistic (Steel et al., 1997, p. 24). Each autocorrelated data value, however, brings less than a full degree of freedom, and thus the effective sample size, which is in the denominator of Equation 3.10, is not n but rather a value less than that.

3.5.2 Modeling Spatial Contiguity

The results of the previous section indicate that the positive autocorrelation among the random variables Y_i in the time series of Equation 3.13 has the effect of increasing the variance $\text{var}\{\bar{Y}\}$. It is reasonable to expect that the same effect may prevail when the random variables are spatially autocorrelated.

The simplest spatial analog for Equation 3.12 is a spatially autocorrelated random variable on a square $m \times m$ lattice. By direct analogy with Equation 3.12 one can write

$$\begin{aligned} Y_i &= \mu + \eta_i \\ \eta_i &= \lambda\left(\sum_{j=1}^{n} w_{ij}\eta_j\right) + \varepsilon_i,\ i = 1,2,\ldots,n. \end{aligned} \tag{3.19}$$

Here $n = m^2$, w_{ij} measures connection strength between Y_i and Y_j, $\varepsilon_i \sim N(0,\sigma^2)$, and as before λ measures the overall autocorrelation strength.

By analogy with time series models, in which one speaks of a time lag, the term $\Sigma w_{ij}\eta_j$ is referred to as a *spatial* lag (Anselin and Bera, 1998). As an example, consider the case $m = 3$. The lattice for this case is shown in Figure 3.7. The matrix $W = [w_{ij}]$ in Equation 3.19 is called the *spatial weights matrix*. One simple W matrix can be constructed by letting

$$w_{ij} = \begin{cases} 1 \textit{ if the cells share a boundary of length} > 0 \\ 0 \textit{ otherwise} \end{cases} \tag{3.20}$$

The matrix W as defined in Equation 3.20 describing the lattice of Figure 3.7 is

$$W = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}, \tag{3.21}$$

as can be verified by inspection. Note that the diagonal elements w_{ii} are all zero. That is, a cell does not share a spatial connection with itself. This is the usual convention in defining the spatial weights matrix.

The spatial weights matrix W determines how the spatial relationships between spatial objects (polygons or points) are modeled. It is evident that the assignment of values to the

1	2	3
4	5	6
7	8	9

FIGURE 3.7
Numbering of a square 3 × 3 lattice.

components w_{ij} of W plays an important role in the modeling of the spatial structure of the data and therefore in the statistical analysis of this structure. There are two aspects of this assignment of values to consider: determining which if any of the w_{ij} are assigned a value of zero, indicating no spatial connection; and determining what number to assign to the nonzero values of w_{ij}.

There is a considerable body of literature on this issue, with many proposed methods for assigning values. In discussing these methods, it is most convenient to separate them initially into two categories. The first category consists of methods for describing relationships among cells such as those of Figure 1.3b, whose boundaries are explicitly defined, and the second consists of collections of methods for describing relationships among spatial points such as those of Figure 1.3a, for which the cell boundaries are not explicitly specified. We discuss four methods of determining the w_{ij} for data falling into the first category, the explicitly defined mosaic of cells. These methods are the rook's case, queen's case, distance, and a method proposed by Cliff and Ord (1981). We will then discuss two methods for determining the w_{ij} for the second category of spatial data, the set of points with no explicitly defined boundaries. These are Euclidean distance and distance threshold.

Beginning with the case of explicitly defined boundaries, the first and simplest choice for W is to assign a positive value to contiguous cells and a value of zero to non-contiguous cells. This requires a definition of contiguity. Probably the most common definition is that cells are contiguous only if they are actually in contact with each other (this is called *first order contiguity*). The *rook's case* contiguity rule specifies that the elements of W are nonzero only for those w_{ij} representing cells whose adjoining boundaries have length greater than zero. In the *queen's case*, those elements of W representing cells whose corners are touching are also assigned nonzero values. The connection to the allowable moves of chess pieces is apparent (to chess players). For the benefit of non-chess players, the rook can only move laterally or forward and back, so that a rook located in lattice cell 5 of Figure 3.7 could only move in the directions of cells 2, 4, 6, and 8. A queen can also move diagonally, so the queen could also move in the direction of cells 1, 3, 7, and 9. In general, we will use the term *rook's case* to refer to any system of assigning values to W in which only boundaries with length greater than zero have nonzero weight, and we will use the *queen's case* to refer to any system of assigning weights in which all boundaries between adjacent cells, whether of zero or greater length, have nonzero values.

When values are assigned according to either rook's case or queen's case contiguity, the simplest rule for assigning a numerical value is to let $w_{ij} = 1$ for contiguous cells and $w_{ij} = 0$ for noncontiguous cells. This is called the *binary* weights assignment. While this has the virtue of simplicity and does simplify certain computational formulas, there is an alternative assignment rule that is also commonly used. This is to assign positive values to contiguous cells in such a way that the row sum $\Sigma_j w_{ij}$ always equals 1. This coding of this weights matrix, called *row normalized*, may be represented as

$$w_{ij} \begin{cases} > 0 \text{ if the cells share a boundary of length} > 0 \\ = 0 \text{ otherwise} \end{cases}$$
$$\sum_{j=0}^{n} w_{ij} = 1. \tag{3.22}$$

With this coding, the W matrix of Figure 3.7 has the form

$$W = \begin{bmatrix} 0 & 1/2 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 \\ 0 & 1/4 & 0 & 1/4 & 0 & 1/4 & 0 & 1/4 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 \end{bmatrix}, \tag{3.23}$$

so that all rows sum to 1. Although this rule does not have a strong intuitive justification, it turns out to have some important advantages (Anselin and Bera, 1998, p. 243). A disadvantage, both conceptually and computationally, is that under this rule W is generally not symmetric, that is, for many combinations of i and j $w_{ij} \neq w_{ji}$. However, every row normalized matrix is *similar* to a symmetric matrix (Ord, 1975), that is, they have the same eigenvalues (Appendix A.1). It is often the case that this property is all that is required to carry out computations associated with spatial analysis. Because of its statistical properties, and because certain operations are greatly simplified by it, we will frequently use this rule in our analyses. There are other rules for assignment besides binary and row normalized; references for these are given in Section 3.7.

In this book, we will only develop models for spatial contiguity using either the rook's case or queen's case rules. However, for purposes of exposition we describe two other rules. These address a problem with the rook's case and queen's case rules, which is that they are based strictly on topological properties, with no reference to geometry. Topological properties are those properties that are invariant under continuous transformation of the map. That is, they are properties that do not change when the map is rotated and distorted as if it were made of rubber. The fact that two polygons share a boundary is a topological property. Geometric properties do vary with continuous distortion; the property that one polygon is north of another, or larger than another, is a geometric property.

Cliff and Ord (1981, p. 15), citing Dacey (1965), refer to the problem with the rook's case and queen's case rules as the problem of "topological invariance." That is, the strength of connection is determined only by the topology of the map, and not by either its geometry or by other factors that might influence the strength of connection between polygons. The potential influence of geometry is shown in Figure 3.8. Under rook's case, contiguity the lattice in Figure 3.8a has the same spatial weights matrix as that in Figure 3.8b. The two figures are not topologically equivalent, but they could be made so by poking a very small hole in the center of the lattice of Figure 3.8a. In this case, the two figures would have the same spatial weights matrix under queen's case contiguity as well. More generally, factors that might influence the degree of contiguity between two polygons include barriers. For example, two polygons that are depicted as contiguous on a map might actually be separated by a river or highway that renders direct transit between them difficult.

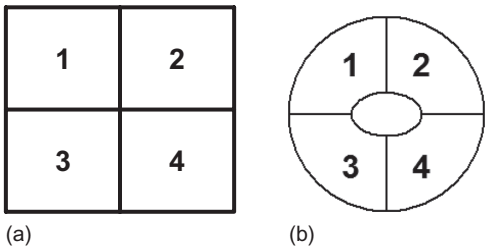

FIGURE 3.8
Under the rook's case contiguity rule, maps (a) and (b) have the same spatial weights matrix. (Modified from Cliff and Ord, 1991, Figure 1.6, p. 15. Used by permission of Pion Limited, London, UK.)

One fairly simple way to define contiguity geometrically is to let $w_{ij} = d_{ij}$, where d_{ij} is some measure of the distance between cell i and cell j. A common choice is to let d_{ij} be the Euclidean distance between centroids of the cells. While this rule does have a certain intuitive appeal, it is not necessarily the best choice. Chou et al. (1990) compared the results of statistical tests in which weights matrices based on this distance rule were used with tests using weights matrices based on some topological form of contiguity rule such as rook's or queen's case. They found that the tests based on the contiguity rules were much more powerful. The tests were carried out on data from an irregular mosaic with greatly varying cell sizes, and the authors' intuitive explanation for the difference in power is that larger cells, which tend to have reduced w_{ij} values due to the greater distance of their centroids from those of other cells, in reality tend to exert the greatest influence on the values of neighboring cells. Manly (1997) points this out as well, and suggests using the inverse of the distance d_{ij}^{-1} instead. Cliff and Ord (1981) suggest the use of the rule $w_{ij} = b_{ij}^{\beta}/d_{ij}^{\alpha}$, where b_{ij} is the fraction of the common border between cell i and cell j that is located in the perimeter of cell i, and α and β are parameters that are chosen to fit the data. This rule would better distinguish between the structures in Figure 3.8a and b. As we have said, however, in this book we will employ only the rook's case or queen's case rules for polygonal data.

We turn now to the specification of weights for the second type of spatial data, in which the location data consist of a finite number of points in the plane as in Figure 1.3a. Since there are no explicit boundaries, a specification by topology, such as the rook's case or queen's case, is impossible. The only means available for specification of weights is geometric. One possibility is to use some form of distance metric such as the inverse distance discussed in the previous paragraph. A second method, which is the one we will use, is to establish a geometric condition that, if satisfied, defines two locations i and j to be neighbors, in which case the value of w_{ij} is greater than zero. The nonzero values of w_{ij} in this formalism are determined similarly to those of the topologically defined neighboring lattice cells already discussed, for example, using a definition analogous to Equation 3.20 or 3.22. In this context, one can either make the condition for a nonzero w_{ij} depend on distance between points being less than some threshold value, or one can declare the k closest points to point i to be neighbors of this point. When the locations of the data form a rectangular set of points such as that defined by the centers of the cells in Figure 1.3b, the threshold distance metric, for an appropriately chosen threshold, generates a weights matrix identical to that of the rook's case (Equation 3.20, see Exercise 3.3). The k nearest neighbors approach does not, because points near the boundary are assigned more neighbors than would be the case under a corresponding lattice or mosaic topology.

3.5.3 Modeling Spatial Association in R

Spatial association is modeled in R by means of the spatial weights matrix W discussed in Section 3.5.2. The representation of this matrix is broken into two parts, matching the two part process of defining the matrix described in that subsection. The first part identifies the matrix elements that are greater than zero (signifying spatial adjacency), and the second assigns a numerical value to adjacent elements. Because spatial weights matrices tend to be very large and sparse (i.e., the vast majority of their terms are zero, as can be seen in Equations 3.21 and 3.23), W is not stored explicitly as a matrix in the `spdep` package. Instead, it is represented by a *neighbor list*, also called an `nb` object, which can be converted into a spatial weights matrix. A neighbor list describing a rectangular lattice like the one in Figure 3.7 can be generated using the `spdep` function `cell2nb()`. The sequence of commands is as follows.

```
> library(spdep)
> nb3x3 <- cell2nb(3,3) #Rook's case by default
# Examine the structure of the neighbor list
> str(nb3x3)
List of 9
 $ : int [1:2] 2 4
 $ : int [1:3] 1 3 5
 $ : int [1:2] 2 6
 $ : int [1:3] 1 5 7
 $ : int [1:4] 2 4 6 8
 $ : int [1:3] 3 5 9
 $ : int [1:2] 4 8
 $ : int [1:3] 5 7 9
 $ : int [1:2] 6 8
 - attr(*, "class")= chr "nb"
 - attr(*, "call")= language cell2nb(nrow = 3, ncol = 3)
 - attr(*, "region.id")= chr [1:9] "1:1" "2:1" "3:1" "1:2" ...
 - attr(*, "cell")= logi TRUE
 - attr(*, "rook")= logi TRUE
 - attr(*, "sym")= logi TRUE
```

The order of cell indices is left to right and top to bottom as in Figure 3.7, so neighbors of cell 1 are cells 2 and 4, and so forth. The argument `type` of the function `cell2nb()` can be used to specify rook's case or queen's case contiguity. Since neither is specified in the listing, the default, which is rook's case, is used. The neighbor list does not specify the numerical values of the nonzero spatial weights.

Assignment of weights is accomplished by the function `nb2listw()`, which creates a `listw` object by supplementing the neighbor list object with spatial weight values. The function `nb2listw()` has an argument `style`, whose value can take on a number of character values, including `"B"` for binary, `"W"` for row normalized, and several others not described in this book. The default value is `"W"`, which is invoked in the example here since no value is given. Only a few lines of the response to the call to function `str()` are shown, but these indicate that the elements of `W33` match those in Equation 3.23.

```
> W33 <- nb2listw(nb3x3)
> str(W33)
List of 3
 $ style : chr "W"
```

```
 $ neighbours:List of 9
  ..$ : int [1:2] 2 4
   *   *   *   DELETED   *   *   *
$ weights:List of 9
  ..$ : num [1:2] 0.5 0.5
  ..$ : num [1:3] 0.333 0.333 0.333
   * * * DELETED   *   *   *
- attr(*, "call")= language nb2listw(neighbours = nb3x3)
```

Let us now use the `raster` and `sp` objects to simulate the spatial autocorrelation in Equation 3.19. Recall that in the case of the temporal model of Equation 3.12, the autocorrelation coefficient λ was required to satisfy $-1 < \lambda < 1$ to keep the terms Y_i from blowing up. The analogous restriction in the case of Equation 3.19 depends on the form of the spatial weights matrix W. For the binary form of Equation 3.20, the restriction is $-¼ < \lambda < ¼$. To see why, note that if $\lambda > ¼$ in Equation 3.19, then the sum of terms on the right-hand side of Equation 3.19 will tend to be larger than the left-hand side, which will cause the Y_i to increase geometrically in magnitude. For the row standardized coding of Equation 3.22, the corresponding restriction is $-1 < \lambda < 1$.

The spatial model (Equation 3.19) is easier to deal with if the equations are written in matrix notation (Appendix A.1). Let Y be the vector of the Y_i (i.e., $Y = [Y_1\ Y_2\ \ldots.\ Y_n]'$ where the prime denotes transpose), and similarly let μ be a vector, each of whose components is the scalar μ, let η be the vector of the η_i (i.e., $\eta = [\eta_1\ \eta_2\ \ldots.\ \eta_n]'$), and similarly let ε be the vector of the ε_i. The second of equations (Equation 3.19) can be written

$$\begin{aligned}
\eta_1 &= \mu + \lambda\left(\sum_{j=1}^{n} w_{1j}\eta_j\right) + \varepsilon_1 \\
\eta_2 &= \mu + \lambda\left(\sum_{j=1}^{n} w_{2j}\eta_j\right) + \varepsilon_2 \\
&\ldots. \\
\eta_n &= \mu + \lambda\left(\sum_{j=1}^{n} w_{nj}\eta_j\right) + \varepsilon_n.
\end{aligned} \tag{3.24}$$

Thus equations (Equation 3.19) may be written in matrix form as

$$\begin{aligned}
Y &= \mu + \eta \\
\eta &= \lambda W\eta + \varepsilon.
\end{aligned} \tag{3.25}$$

The term $W\eta$ is the spatial lag in this case (Anselin, 1988, p. 22). However, because the lag is applied to the error in Equation 3.25, this particular model is called the *spatial error model* (see Chapter 13).

Haining (1990, p. 116) describes the following method of simulating this model. Subtracting $\lambda W\eta$ from both sides of the second of equations (Equation 3.25) yields $\eta - \lambda W\eta = \varepsilon$, or $(I - \lambda W)\eta = \varepsilon$. This equation may be inverted to obtain

$$\eta = (I - \lambda W)^{-1}\varepsilon. \tag{3.26}$$

We can generate a vector $Y = \{Y_1, Y_2, ..., Y_n\}$ of autocorrelated random variables by generating a vector of $\varepsilon = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_n\}$ of normally distributed pseudorandom numbers, premultiplying μ by the matrix $(I - \lambda W)^{-1}$ to obtain η, and adding μ in Equation 3.25 to obtain Y. The R package `spdep` implements this solution method. We will demonstrate it using a square grid generated by the function `cell2nb()`. Because cells on the boundary of the lattice only have a neighbor on one side, they have asymmetric interactions. This can cause a non-stationarity analogous to the initial transient in the time series model of Equation 3.12, discussed in Section 3.5.1. In that example, the first ten values of Y_i were discarded to eliminate or at least reduce initial transient effects. In a similar way, we will reduce the boundary effect by generating a larger region than we intend to analyze and deleting the values Y_i of lattice cells near the boundary. We will generate code to implement equation (Equation 3.25) with $\mu = 0$ (so that the null hypothesis is true) for a square grid of 400 cells plus two cells on the boundary that will be dropped. Since $\mu = 0$, we have $Y = \eta$, and therefore equations (Equation 3.25) become $Y = \lambda WY + \varepsilon$.

In code below the side of the region actual region computed is set at 24 and then two cells are removed from each boundary to create the final display. The grid cells are placed at the center of cells running from 0 to 24.The function `expand.grid()` (see Exercise 3.4) is used to generate a data frame `Y.df` of coordinates of the centers of the lattice cells.

```
> library(spdep)
> Y.df <- expand.grid(x = seq(0.5,23.5),
+   y = seq(23.5, 0.5, by = -1))
```

Notice the second argument to the function `expand.grid()`, which is `y = seq(23.5, 0.5, by = -1)`. This sequences y down from 23.5 to 0.5 and ensures that the generated values of y are matched with the correct grid cell. The next step is to create a data field `Y` in `Y.df` with the implementation of Equation 3.26. First we assign a value to λ and create the square grid. This is again done using `cell2nb()`. Given the neighbor list created in this way, the function `invIrM()` generates the matrix $(I - \lambda W)^{-1}$.

```
> lambda <- 0.4
> nlist <- cell2nb(24, 24)
> IrWinv <- invIrM(nlist, lambda)
```

Since no value of `type` is specified in the arguments `cell2nb()`, the default rook's case neighbor list is generated. Similarly, since no `style` argument is specified in the arguments of `invIrM()`, the default, which is row normalized spatial weights, is used. The R operator for matrix multiplication (Appendix A.1) is `%*%`, so that the generation of the vector λ and multiplication by $(I - \lambda W)^{-1}$ is carried out as follows.

```
> eps <- rnorm(24^2)
> Y.df$Y <- IrWinv %*% eps
```

The final step is to remove the boundary cells from the lattice.

```
> Y.df <- Y.df[(Y.df$x > 2 & Y.df$x < 22
+  & Y.df$y > 2 & Y.df$y < 22),]
```

Figure 3.9 contains thematic maps of the values of the spatial lattices created using a grid of raster cells whose attribute values are the data generated above using `expand.grid()`. The figures show the spatial patterns if Y for three values of the autocorrelation parameter λ. The pattern for $\lambda = 0.8$ appears to have more of a tendency for high values to

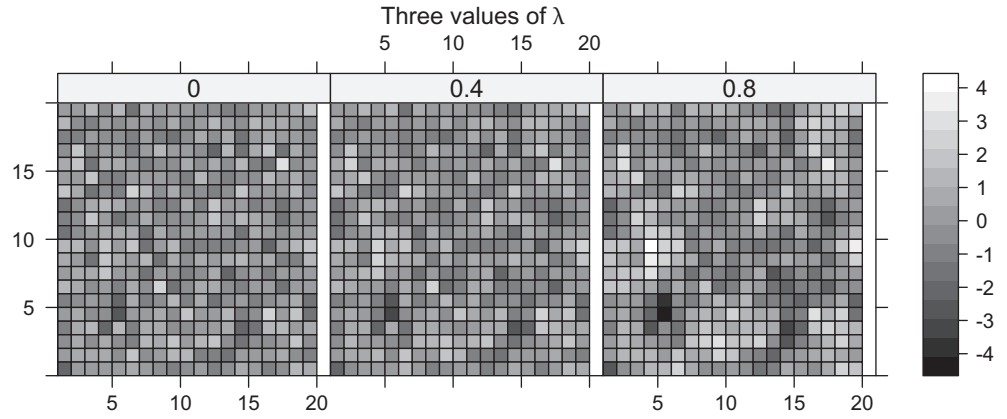

FIGURE 3.9

Gray-scale maps of values of Y in a spatial autoregression model for three values of λ.

be near other high values and low values to be near low values than does the pattern for $\lambda = 0$. As these figures show, however, it is not always easy to identify the level of spatial autocorrelation by eye.

The construction of the grids in Figure 3.9 provides an opportunity to discuss some aspects of the use of spatial objects from the `raster` and `sp` classes and of the process of coercion. If you are not too familiar with these classes, it would be a good idea to review them in Section 2.4.3. The three sets of values of `Y.df` computed above are defined on a square 20 by 20 grid. We can create such a grid using the creation function `raster()`. This was used in Section 2.4.3 to read a file from the disk, but it can also be used to create a grid directly.

```
> library(raster)
> Y.ras <- raster(ncol = 20, nrow = 20, xmn = 0, xmx = 20, ymn = 0,
+   ymx = 20,crs = NULL)
```

This creates a 20 by 20 `rasterLayer` grid. By default the grid has WGS84 coordinates, and the argument `crs = NULL` suppresses this. The next step is to use the coercion function `as()` to coerce this into a `SpatialPolygons` object.

```
> Y.sp <- as(Y.ras, "SpatialPolygons")
```

A `SpatialPolygons` object is, in GIS terminology, an object that has spatial data but no attribute data (Section 2.4.3). The attribute data is contained in the data frame `Y.df` created above. We use the constructor function `SpatialPolygonsDataFrame(Sp, data, match.ID)` to create the `SpatialPolygonsDataFrame` object `Y.spdf`.

```
> Y.spdf <- SpatialPolygonsDataFrame(Y.sp, Y.df, FALSE)
```

The first argument specifies the `SpatialPolygons` object, the second argument specifies the data frame, and the third argument specifies whether the attribute data records are to be matched with the polygons by matching their row names. In our case, these row names are different, and so it is vitally important to verify that the order of the data records in `Y.df` is the same as the order of the polygons in `Y.sp` (Exercise 3.4). This is sufficiently important that we will devote a subsection to it in Section 3.6 when we work with real data, and as preparation for this you should do the exercise.

In order to examine the effect of spatial autocorrelation on the test of the null hypothesis $\mu = 0$ we will use functions from the `spdep` package to carry out a Monte Carlo simulation experiment. Formally, the null and alternative hypotheses are the same as Equation 3.6,

$$\begin{aligned} H_0 &: \mu = 0, \\ H_a &: \mu \neq 0, \end{aligned} \tag{3.27}$$

where μ is a parameter in Equation 3.25. In the simulations, the value of μ is set to zero so that the null hypothesis is true. The results of Monte Carlo experiments in Section 3.4.1 indicated that an increase in the value of the autocorrelation term λ in a time series led to an increase in the Type I error rate. We will see whether the same phenomenon is observed in the case of spatially autocorrelated data by carrying out a series of Monte Carlo simulations of hypothesis tests of models of the form (Equation 3.25) with increasing values of λ. The code for these tests does not require that the locations of the lattice cells be specified explicitly; these locations are implicit in the spatial weights matrix W describing them. The coding is therefore slightly different from that just given. We use `cell2nb()` to generate a neighbor list of a 14 by 14 grid of cells, generate the random numbers and then remove the outer two cells to remove edge effects. Then the function `invIrM()` is applied to the neighbor list, again with the default values. We again use the defaults to create a row normalized rook's case spatial weights matrix.

```
> lambda <- 0.4
> nlist <- cell2nb(14,14)
> IrWinv <- invIrM(nlist, lambda)
```

Since we will use the function `replicate()` to run the simulations, we must create a function that we call `ttest()` to generate each individual simulation.

```
> ttest <- function(IrWinv){
+ # Generate a vector representing the autocorrelated data
+ # on a 14 by 14 grid
+   Y.plus <- IrWinv %*% rnorm(14^2)
+ # Convert Y to a matrix and remove the outer two cells
+   Y <- matrix(Y.plus, nrow = 14,
+      byrow = TRUE)[3:12,3:12]
+   Ybar <- mean(Y)
+ # Carry out the test and return the outcome and the mean
+   t.ttest <- t.test(Y,alternative = "two.sided")
+   TypeI <- t.ttest$p.value < 0.05
+   return(c(TypeI, Ybar))
+ }
```

Here are the results of the simulation.

```
> set.seed(123)
> U <- replicate(10000, ttest(IrWinv))
> mean(U[1,])
[1] 0.1876
> mean(U[2,])
[1] -0.0009136907
> sqrt(var(U[2,]))
[1] 0.1594549
```

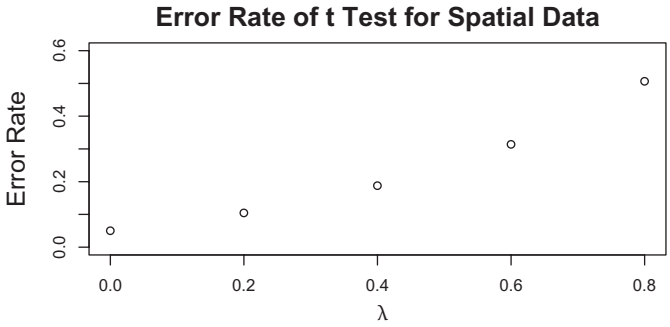

FIGURE 3.10
Type I error rate as a function of λ in 10,000 Monte Carlo simulations of a spatial autoregression model.

In the listing, the autocorrelation term `lambda` is set to 0.4, and the error rate is inflated.

Figure 3.10 shows the Type I error rate as a function of the value of λ used in the simulation. At the value $\lambda = 0$, the error rate is approximately 0.05. As in the case of time series, the error rate increases with increasing λ. The spatial analog of Equation 3.17, describing the effects of autocorrelation on the variance of $\bar{Y}$, is

$$\begin{aligned} \text{var}\{\bar{Y}\} &= \text{var}\left\{\frac{1}{n}\sum_i Y_i\right\} \\ &= \frac{1}{n^2}\sum_i \text{var}\{Y_i\} + \frac{2}{n^2}\sum_{i\neq j}\text{cov}\{Y_i, Y_j\} \\ &= \frac{\sigma^2}{n} + \frac{2}{n^2}\sum_{i\neq j}\text{cov}\{Y_i, Y_j\}. \end{aligned} \tag{3.28}$$

and a similar analog exists for Equation 3.18, describing the effect of autocorrelation on $E\{\sigma^2\}$. As with temporally autocorrelated data, so with spatially autocorrelated data the t statistic is larger than it would be if the random variables Y_i were uncorrelated, so that the Type I error rate is inflated. To repeat, when data are spatially autocorrelated, each data value provides some information about the other data values near it, so that the effective sample size less than the number n of data records.

3.6 Application to Field Data

3.6.1 Setting Up the Data

Our primary descriptor of spatial relationships is the spatial weights matrix. The elements of this matrix depend on the rules used to define what it means to be a spatial neighbor. The choices for neighbor status on which we focus are the rook's case and queen's case for polygons (Figure 1.3b); and threshold distance and k nearest neighbors for points (Figure 1.3a). In this section we examine, using a real data set, the sensitivity

of the results of spatial analyses to the way this matrix is constructed. The data we will use are detrended percent sand content in Field 1 of Data Set 4 (abbreviated Field 4.1). The data, detrended using linear regression and median polish, are shown in Figure 3.4. These figures indicate that the detrended sand content data may be spatially autocorrelated in this field. Both methods tend to leave low detrended values near the center and higher values in the northern and southern ends.

We will fit the spatial error model of Equation 3.25 to the sand data that have been detrended using the linear model (Figure 3.4b). There are other models besides the spatial error model that may fit these data better, and no attempt is made at this point to justify the use of the model; this issue is taken up in Chapter 13. For the present, we simply use this as an example of one type of model that might be used for spatially autocorrelated data. We first test methods for modeling spatial relationships between point data such as those displayed in Figure 3.4b. The first method that we will use to generate the nonzero values of a spatial weights matrix is based on a threshold distance between neighboring points, and the second is based on selecting the k nearest neighbors of each point. We begin with the former. We identify the locations using the UTM coordinates, which are in the data fields *Northing* and *Easting*. Because these are large numbers that, when squared, disrupt the accuracy of the linear regression, we first create new coordinates by subtracting the minimum values. The input of the data frame `data.Set4.1` is described in Appendix B.4.

```
> data.Set4.1$x <-data.Set4.1$Easting - min(data.Set4.1$Easting)
> data.Set4.1$y <-data.Set4.1$Northing - min(data.Set4.1$Northing)
```

Next, we fit the trend surface and use the function `predict()` described in Section 3.2.1 to create this surface. The trend is subtracted from the `Sand` data field to create the detrended sand content `SandDT`.

```
> trend.lm <- lm(Sand ~ x + y + I(x^2) +
+   I(y^2) + I(x*y), data = data.Set4.1)
> data.Set4.1$SandDT <- data.Set4.1$Sand - predict(trend.lm)
```

Next, the data frame is converted into a spatial points data frame using the function `coordinates()`. The function `dnearneigh()` from the `spdep` package is used to create a neighbors list in which points within 61 m (the point to point distance of the sample locations in the field) are considered neighbors.

```
> coordinates(data.Set4.1) <- c("x", "y")
> nlist <- dnearneigh(data.Set4.1, d1 = 0, d2 = 61)
```

The neighbor list is used to construct a `listw` object representing the spatial weights matrix, and then the function `errorsarlm()` is applied to generate the estimated value of λ in Equation 3.25.

```
> W <- nb2listw(nlist, style = "W")
> Y.mod <- errorsarlm(SandDT ~ 1, data = data.Set4.1, listw = W)
> print(Y.mod$lambda, digits = 4)
   lambda
0.4005
```

The estimate is $\hat{\lambda} = 0.4005$. When this operation is repeated using the argument `style = "B"` in the first line, the estimate is $\hat{\lambda} = 0.1180$. Recall that the limiting values of λ for stationarity

are $-1 < \lambda < 1$ for the binary W and $-1/4 < \lambda < 1/4$ for the binary W, so we expect the estimated value $\hat{\lambda}$ for the binary W to be about one-fourth that of the row normalized W.

Next, we try an implementation of the k nearest neighbors rule using the function `knearneigh()`. For a rectangular grid such as this, the four nearest neighbors are the only locations within the point to point distance threshold. Note that this function does not generate the neighbor list directly and has to be called as an argument of the function `knn2nb()`.

```
> nlist <- knn2nb(knearneigh(data.Set4.1, k = 4))
> W <- nb2listw(nlist)
> Y.mod <- errorsarlm(SandDT ~ 1, data = data.Set4.1, listw = W)
> print(Y.mod$lambda, digits = 4)
   lambda
0.3299
```

The estimate is $\hat{\lambda} = 0.3299$. The estimate for binary spatial weights is $\hat{\lambda} = 0.0824$. In this example there is a considerable difference, possibly due to the difference between the weights matrices for points on the boundary.

One of the questions that will be of interest is the extent to which the method used to detrend the data influences the results. Referring to Equation 3.1, $Y(x,y) = T(x,y) + \eta(x,y) + \varepsilon(x,y)$, different detrending methods will apportion the values of $Y(x,y)$ in this equation to the three components on the right-hand side in different ways. To repeat the comment of Cressie, "one person's deterministic trend may be another person's correlated error structure" (by the end of this book you will be tired of reading this). In Exercise 3.8, you are asked to repeat these calculations for data that have been detrended using median polish and for data that have not been detrended at all.

We turn now to polygonal data. Since the data themselves are point measurements, the creation of polygons to represent these data is somewhat arbitrary, particularly since the field is not rectangular (Figure 3.4). The most natural choice is to create Thiessen polygons. Given a set of points $P_1, P_2, \ldots, P_n$ in the plane, Thiessen polygons (also called Voronoi polygons or Dirichlet cells) have the property that every polygon contains one of the points P_i as well as every point in the region that is closer to P_i than to any other point in the set $P_1, P_2, \ldots, P_n$ (Ripley, 1981, p. 38; Lo and Yeung, 2007, p. 333). When the set of points forms a regular grid, Thiessen polygons form a square or rectangular lattice. R has the capacity to construct Thiessen polygons using functions in the packages `tripack` (Renka et al., 2011), `deldir` (Turner, 2011), and `spatstat` (Baddeley and Turner 2005).

We will first use the point pattern analysis package `spatstat` to create a `ppp` file, which is a `spatstat` point format representing a point pattern data set in the two-dimensional plane. This file is created using the constructor function `ppp()`, which takes as arguments a vector of the x coordinates of the points, a vector of the y coordinates, and an argument `owin` that defines the bounding box (see Section 2.4.3) of the region.

```
> library(spatstat)
> W <- 592000
> E <- 592500
> S <- 4270300
> N <- 4271200
> cell.ppp <- ppp(data.Set4.1$Easting, data.Set4.1$Northing,
+       window = owin(c(W, E), c(S, N)))
```

Next, we use the `spatstat` function `dirichlet()` to create a `spatstat` object of class `tess` (for "tessellation").

```
> thsn.tess <- dirichlet(cell.ppp)
```

Next, we follow the process we used in Section 3.5.3. We use a coercion function from `maptools` to coerce the `tess` object into a `SpatialPolygons` object using the coercion function `as()`. Then we use the constructor function `SpatialPolygonsDataFrame()` to create the `SpatialPolygonsDataFrame` object `thsn.spdf`.

```
> library(maptools)
> thsn.sp <- as(thsn.tess, "SpatialPolygons")
> thsn.spdf <- SpatialPolygonsDataFrame(thsn.sp,
+   slot(data.Set4.1, "data"), FALSE)
```

Before we can move on, we must make sure that the ordering of the spatial data matches the ordering of the attribute data. That is the subject of the next subsection.

3.6.2 Checking Sequence Validity

In the last section and in Exercise 3.4, we alluded to an issue that is sufficiently subtle, and whose consequences are sufficiently dire, that it is worthwhile to emphasize it by devoting an entire subsection. Look at the two lines of code just above. The first line creates the object `thsn.sp`, which describes the polygon structure and contains spatial data but no attribute data. It was created by forming Thiessen polygons around the set of point coordinates in `data.Set4.1`, and these polygons are arrayed in an order determined by the creation process, independent of the values in the `ID` field in `data.Set4.1`. Now look at the following line of code. This assigns attribute data to the polygons. There is no guarantee that the order of the attribute data matches the order of the polygons, and if it does not, then the attribute data will be assigned to the wrong polygons. Therefore, in these situations one must always check to make sure that the polygon order matches the attribute data order.

Because it is simpler to work with `sf` (spatial features) objects, we will use coercion to move into this domain.

```
> library(sf)
> thsn.geom.sf <- st_as_sf(thsn.sp)
> str(thsn.geom.sf)
Classes 'sf' and 'data.frame':      86 obs. of  1 variable:
 $ geometry:sfc_POLYGON of length 86; first list element: List of 1
  ..$ : num [1:5, 1:2] 592081 592081 592000 592000 592081 ...
  ..- attr(*, "class")= chr "XY" "POLYGON" "sfg"
 - attr(*, "sf_column")= chr "geometry"
 - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..:
  ..- attr(*, "names")= chr
```

Although there are no identification numbers, the 86 polygons are arranged in a particular order, namely, the order assigned to them when they were created. Again, we coerce `thsn.spdf` into an `sf` object and look at its structure.

```
> thsn.sf <- st_as_sf(thsn.spdf)
> str(thsn.sf)
```

```
Classes 'sf' and 'data.frame':      86 obs. of  27 variables:
  $ ID        : int 1 2 3 4 5 6 7 8 9 10 ...
  $ Row       : int 1 1 1 1 1 1 1 2 2 2 ...
  $ Column    : int 1 2 3 4 5 6 7 1 2 3 ...
  $ Easting   : num 592051 592112 592173 592234 592295 ...
  $ Northing  : num 4271104 4271104 4271104 4271104 4271104 ...
...
```

Now we come to the important part. The data records in the data frame `data.Set4.1` are indexed by the data field `ID`. The polygons in `thsn.sf` were created based on the values of Easting and Northing in this data frame. Therefore, the values of `thsn.sf$ID` should match the order of records of this data frame. With 86 records this is easy to check.

```
> thsn.sf$ID
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
[23] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
[45] 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
[67] 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
```

If there are hundreds of data records, it might be less easy to visually, but we can do it with the function `all.equal()`.

```
> all.equal(1:length(thsn.sf$ID), thsn.sf$ID)
[1] TRUE
```

We can also check for correct alignment visually. First, we plot the attribute *ID* values.

```
> plot(thsn.spdf, pch = 1, cex = 0.1)
> text(coordinates(thsn.spdf),
+      labels=as.character(thsn.spdf@data$ID))
```

Next, we plot the polygon *ID* values and see that they match.

```
> plot(thsn.spdf, pch = 1, cex = 0.1)
> text(coordinates(thsn.spdf),
+      labels=lapply(thsn.spdf@polygons, slot, "ID"))
```

Now that we have assured ourselves that there is no misalignment of geometry and attribute data, we can move on to the spatial autocorrelation tests. See Exercise 3.8 for some practice on this topic.

3.6.3 Determining Spatial Autocorrelation

There are four cases we will consider in estimating the parameter λ of Equation 3.25 for the Thiessen polygons, namely, the possible combinations of rook's case and queens case contiguity rules and binary and row normalized spatial weights. The function used to carry out the estimation is again `errorsarlm`.

```
> nlist <- poly2nb(thsn.spdf,
+   row.names = as.character(thsn.spdf$ID), queen = FALSE)
> W <- nb2listw(nlist, style = "W")
> Y.mod <- errorsarlm(SandDT ~ 1, data = thsn.spdf, listw = W)
> print(Y.mod$lambda, digits = 4)
   lambda
0.4033
```

The corresponding estimate for the binary rook's case W is $\hat{\lambda} = 0.1182$. For queen's case contiguity, the estimates are $\hat{\lambda} = 0.2009$ for the row normalized W and $\hat{\lambda} = 0.0407$ for the binary W. Thus, with this data set there is little difference between the estimates for different contiguity rules. Not surprisingly, the rook's case contiguity rule gives a similar estimate to that of the corresponding nearest neighbor distance rule for point data. Which estimate is "better"? We don't know if any of them are any good, because the model itself could be in error. We will leave the discussion of this issue until we take up autoregressive models in Chapter 13.

3.7 Further Reading

The fundamental source for almost all of the material discussed in this chapter is Cliff and Ord (1981). The seminal paper on the modeling spatial autocorrelation is generally considered to be that of Whittle (1954). This paper explicitly draws on the analogy of spatial processes with time series, using the notion of a line transect in space as a conceptual bridge. Bartels (1979) also provides a discussion of the relationship between time series data and spatial data.

Davis (1986) provides an extensive discussion of the properties of trend surfaces and develops a method for testing their significance. The difficulty in distinguishing between responses to a trend and a spatial error structure is amplified by Sokal et al. (1993). They point out that apparent spatial autocorrelation may be due to the response of the mean to a deterministic regional trend. They refer to this as "spurious spatial autocorrelation."

There are a number of references that provide a more detailed discussion of creating a spatial weights matrix W (Cliff and Ord, 1981; Anselin, 1988; Anselin and Bera, 1998; Tiefelsdorf, 2000). Getis and Aldstadt (2004) describe a method for creating W using local spatial statistics, which are discussed in Section 4.5.3. Several papers discuss the assignment of spatial weights in various contexts. For example, Tiefelsdorf et al. (1999) describe the effects of various choices for the structure of W on the exact distribution of Moran's I. Case and Rosen (1993) use a combination of spatial and attribute data to assign weights between states to model budget development. Lacombe (2004) analyzes different methods for generating the matrix in the context of county welfare policy.

All of the artificial spatial data sets in this book are generated using the method of Haining (1990, p. 116), which is simple and easy to understand. This is not, however, the only way to generate random spatial data. Griffiths (1988, Ch. 9) provides an extensive discussion of the simulation of spatially autocorrelated data. The `geoR` package (Ribeiro and Diggle, 2016) contains the function `grf()`, which develops a Gaussian random fields simulation of autocorrelated data (Wood and Chan, 1994).

Finally, G. E. P. Box's more famous quote is "all models are wrong; some models are useful," which is from Box (1979).

Exercises

3.1 To test the sensitivity of the t-test to skewed data, carry out a Monte Carlo simulation of the test using data generated from a normal and a lognormal distribution. Be careful about the value of μ for the lognormal distribution.

3.2 The Monte Carlo simulation should not be applied blindly. Consider the coin tossing experiment conducted in Section 3.2.2. Look up the function `binom.test()` and then carry out a simulation of 50 tosses of a fair coin using the following function:

```
coin.toss <- function (n.tosses){
  z <- rbinom(1,n.tosses,0.5)
  n.heads <- sum(z)
  b <- binom.test(n.heads, n.tosses,0.5,"two.sided",0.95)
  TypeI <- b$p.value < 0.05
}
```

Use the function `replicate()` to run this function 10,000 times and compute the fraction of Type I errors. Is it close to 0.05? Why not? HINT: explore the possible values that `b$p.value` can take on.

3.3 Using the `spdep` functions described in this chapter, generate a `listw` object for a four by four square lattice. Use the function `listw2mat()` to generate a row normalized spatial weights matrix. Use `apply()` to show that the rows sum to 1.

3.4 Use the function `expand.grid()` to generate a three by three set of points matching the lattice of Figure 3.7. Then use the `spatstat` functions discussed in this chapter to generate Thiessen polygons. Use the function `str()` to check whether the numbering of these polygons matches Figure 3.7. How do you have to order the x and y sequences in `expand.grid()`?

3.5 (a) Use the function `read.csv()` to read Data Set 2 (don't forget to set your working directory). From this data set, create a small data set by selecting only those sites with longitude values between –123.2463° and –123.0124° and latitude values between 39.61633° and 39.95469°; (b) Use the statement `?coordinates` to read about the function `coordinates()`. Then use this function to create a `SpatialPointsDataFrame` by assigning data fields to be the coordinates of the data set; (c) Plot the locations of the data set using the `plot()` function; (d) Use the function `str()` to explore the structure of the `SpatialPointsDataFrame`.

3.6 (a) `Use knearneigh()` to create a spatial weights matrix and use the function `errorsarlm()` to estimate the value of λ in a spatial error model for the data set created in Problem 3.5; (b) Give a physical interpretation of the meaning of the terms in the spatial lag model (Equation 3.25) as applied to the data in part (a). Does this model make sense?

3.7 Use Monte Carlo simulation to compute the Type I error rate for a 10 by 10 square data set created with an autocorrelation value of λ equal to 0.6. Compute the sample mean and sample standard deviation of the replicates.

3.8 It may help with the idea of ID misalignment to see what happens when data are misaligned with polygons. Create a data set identical to Data Set 4.1 of Section 3.6.2 but with order of the ID values reversed so that they run from 86 to 1 and repeat the analyses of that section on this data set.