

# 5

## GRAPHS AND MAPS

“The ideal situation occurs when the things that we regard as beautiful are also regarded by other people as useful.”

—Donald Knuth

Graphs and maps help you reason with data. They also help you communicate results. A good graph gives you the most information in the shortest time, with the least ink in the smallest space (Tufte, 1997). In this chapter, we show you how to make graphs and maps using R.

A good strategy is to follow along with an open session, typing (or copying) the code as you read. Before you begin make sure you have the following data sets available in your working directory. Do this by typing

```
> SOI = read.table("SOI.txt", header=TRUE)
> NAO = read.table("NAO.txt", header=TRUE)
> SST = read.table("SST.txt", header=TRUE)
> A = read.table("ATL.txt", header=TRUE)
> US = read.table("H.txt", header=TRUE)
```

Not all the code is shown but all is available on our Web site.

### 5.1 GRAPHS

It is easy to make a graph. Here we provide guidance to help you make informative graphs. It is a tutorial on how to create publishable figures from your data. In R you have several choices. With the standard (base) graphics environment, you can produce a variety of plots with fine details. Most of the figures in this book use the standard graphics environment. The grid graphics environment is even more flexible.

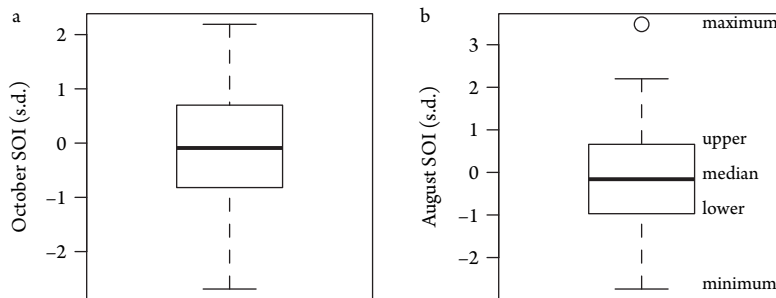


Figure 5.1 Box plot of the October SOI.

It allows you to design complex layouts with nested graphs where scaling is maintained upon resizing. The **lattice** and **ggplot2** packages use grid graphics to create more specialized graphing functions and methods. The `splot` function for example is plot method built with grid graphics that you will use to create maps. The **ggplot2** package is an implementation of the grammar of graphics combining advantages from the standard and lattice graphic environments. It is worth the effort to learn. We begin with the standard graphics environment.

### 5.1.1 Box Plot

A box plot is a graph of the five-number summary. The `summary` function applied to data produces the sample mean along with five other statistics including the minimum, the first quartile value, the median, the third quartile value, and the maximum. The box plot graphs these numbers. This is done using the `boxplot` function. For example, to create a box plot of your October SOI data, type

```
> boxplot(SOI$Oct, ylab="October SOI (s.d.)")
```

Figure 5.1 shows the results. The line inside the box is the median value. The bottom of the box (lower hinge) is the first quartile value and the top of the box (upper hinge) is the third quartile. The vertical line (whisker) from the top of the box extends to the maximum value and the vertical line from the bottom of the box extends to the minimum value.

Hinge values equal the quartiles exactly when there is an odd number of observations. Otherwise, hinges are the middle value of the lower (or upper) half of the observations if there is an odd number of observations below the median and are the middle of two values if there is an even number of observations below the median. The `fiveum` function gives the five numbers used by `boxplot`. The height of the box is the interquartile range (IQR) and the range is the distance from the bottom of the lower whisker to the top of the upper whisker.

By default, the whiskers are drawn as a dashed line extending from the box to the minimum and maximum data values. Convention is to make the length of the whiskers no longer than 1.5 times the height of the box. The outliers, data values

larger or smaller than this range, are marked separately with points. Figure 5.1 also shows the box plot for the August SOI values. The text identifies the values. Here there is a single outlier. In this case, the upper whisker extends to the last data value less than  $1.5 \times \text{IQR}$ .

For example, if you type

```
> Q1 = fivenum(SOI$Aug) [2]
> Q2 = fivenum(SOI$Aug) [3]
> Q3 = fivenum(SOI$Aug) [4]
> Q2 + (Q3 - Q1) * 1.5
[1] 2.28
```

you see one observation greater than 2.3. In this case, the upper whisker ends at the next highest observation value less than 2.3. Observations above and below the whiskers are considered outliers. You can find the value of the single outlier of the August SOI by typing

```
> sort(SOI$Aug)
```

The largest observation in the data less than 2.3 is 2.2.

Your observations are said to be symmetric if the median is near the middle of the box with the two whiskers of equal lengths. A symmetric set of observations will also have the same number of high and low outliers.

Twenty-five percent of all your observations are below the lower quartile (below the box), 50% are below (and above) the median, and 25% are above the upper quartile. The box contains 50% of all your data. The upper whisker extends from the upper quartile to the maximum and the lower whisker extends from the lower quartile value to the minimum except if they exceed 1.5 times the interquartile range above the upper or below the lower quartiles. In this case, outliers are plotted as points. This outlier option can be turned off by setting the *range* argument to zero.

The box plot is an efficient graphical summary of your data. By removing the box lines altogether, the same information is available with less ink. Figure 5.2 is series of box plots representing the SOI for each month. The dot represents the median; the ends of the lines toward the dot are the lower and upper quartiles, respectively; the ends of the lines toward the bottom and top of the graph are the minimum and maximum values, respectively.

### 5.1.2 Histogram

A histogram is a graph of the distribution of your observations. It shows where the values tend to cluster and where they tend to be sparse. The histogram is similar but not identical to a bar plot (see Chapter 2). The histogram uses bars to indicate frequency (or proportion) in data intervals, whereas a bar plot uses bars to indicate the frequency of data by categories. The `hist` function creates a histogram.

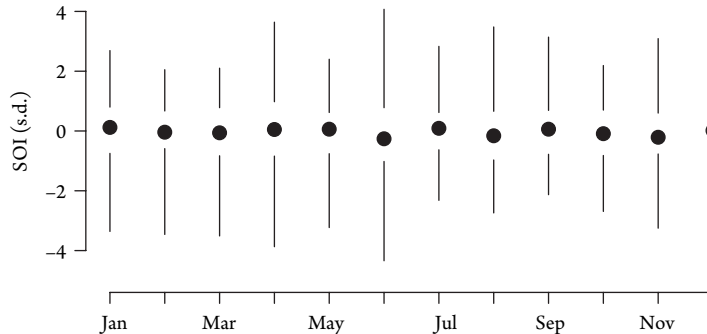


Figure 5.2 Five-number summary of the monthly SOI.

Consider NOAA’s annual values of accumulated cyclone energy (ACE) for the North Atlantic and July SOI values. Annual ACE is calculated by squaring the maximum wind speed for each six-hour tropical cyclone observation and summing over all cyclones in the season. The values obtained from NOAA (<http://www.aoml.noaa.gov/hrd/tcfaq/E11.html>) are expressed in units of knots squared  $\times 10^4$ . You create the two histograms and plot them side by side. First set the plotting parameters with the `par` function. Details on plotting options are given in Murrell (2006). After your histogram is plotted, the function `rug` adds tick marks along the horizontal axis at the location of each observation (like a floor carpet).

```
> par(mfrow=c(1, 2), pty="s")
> hist(A$ACE)
> rug(A$ACE)
> hist(SOI$Jul)
> rug(SOI$Jul)
```

Figure 5.3 shows the result. Here we added an axis label, turned off the default title, and placed text (“a” and “b”) in the figure margins. Plot titles are useful in presentations, but are redundant in publication. The default horizontal axis label is the name of the data vector. The default vertical axis is frequency and is labeled accordingly.

Default values for the `hist` function options provide a good starting point, but you might want to make adjustments. It helps to know how the histogram is assembled. First a contiguous collection of disjoint intervals, called bins (or classes), is chosen that cover the range of data values. The default for the number bins is the value  $\lceil \log_2(n) + 1 \rceil$ , where  $n$  is the sample size and  $\lceil \cdot \rceil$  indicates the ceiling value (next largest integer). If you type

```
> n = length(SOI$Jul)
> ceiling(log(n, base=2) + 1)
[1] 9
```

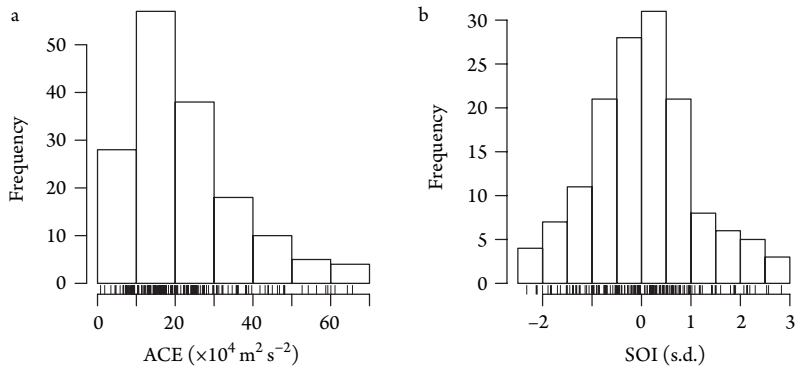


Figure 5.3 Histograms of (a) ACE and (b) SOI.

you can see that adjustments are made to this number so that the cut points correspond to whole number data values. In the case of ACE, the adjustment results in 7 bins and in the case of the SOI it results in 11 bins. Thus the computed number of bins is a suggestion that gets modified to make for nice breaks.

The bins are contiguous and disjoint so the intervals look like  $(a, b]$  or  $[a, b)$  where the interval  $(a, b]$  means from  $a$  to  $b$  including  $b$  but not  $a$ . Next, the number of data values in each of the intervals is counted. Finally, a bar is drawn above the interval so that the bar height is the number of data values (frequency). A useful argument to make your histogram understandable is `prob=TRUE`, which allows you to set the bar height to the density, where the sum of the densities times the bar interval width equals one.

You conclude that ACE is positively skewed with some few years having very large values. By contrast, the SOI appears symmetric with short tails as you would expect from a normal distribution.

### 5.1.3 Density Plot

A histogram outlines the general shape of your data. Usually that is sufficient. You can adjust the number of bins (or bin width) to get more or less detail on the shape. An alternative is a density plot. A density plot captures the distribution shape by smoothing the histogram. Instead of specifying the bin width, you specify the amount (and type) of smoothing. There are two steps. First you use the `density` function to obtain a set of kernel density estimates from your observations. Second you plot these estimates using the `plot` method.

A kernel density is a function that provides an estimate of the average number of values at any location in the space defined by your data. This is illustrated in Figure 5.4, where the October SOI values in the period 2005–2010 are indicated as a rug, and a kernel density function is shown as the black curve. The height of the function, representing the local density, is a sum of the heights of the individual kernels shown in red.

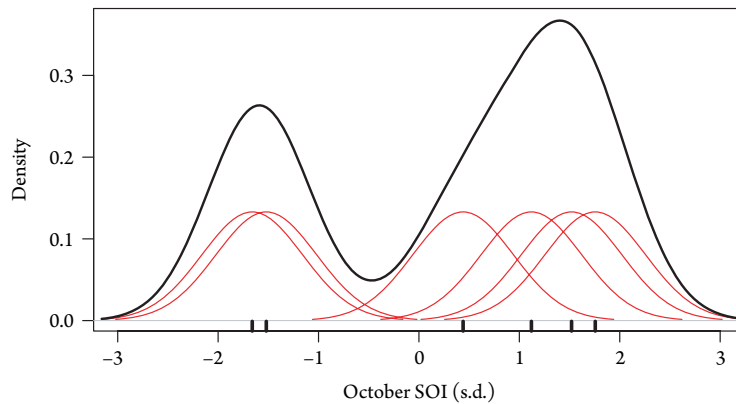


Figure 5.4 Density of October SOI (2005–2010).

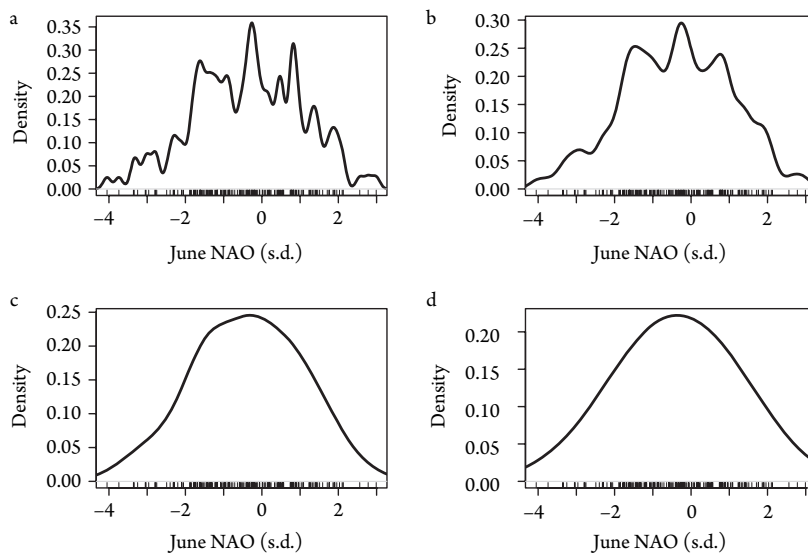


Figure 5.5 Density of June NAO. (a) .1, (b) .2, (c) .5, and (d) 1 s.d. bandwidth.

The kernel is a Gaussian (normal) distribution centered at each data value. The width of the kernel, called the bandwidth, controls the amount of smoothing. The bandwidth is the standard deviation of the kernel in the `density` function. This means the inflection points on the kernel occur one bandwidth away from the data location in units of the data values. Here with the SOI in units of standard deviation, the bandwidth equals .5 s.d.

A larger bandwidth produces a smoother density plot for a fixed number of observations because the kernels have greater overlap. Figure 5.5 shows the density plot of June NAO values from the period 1851 to 2010 using bandwidths of .1, .2, .5, and 1.

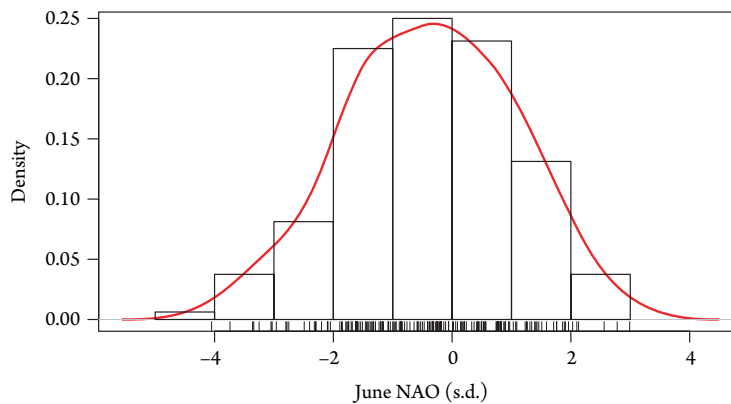


Figure 5.6 Density and histogram of June NAO.

The smallest bandwidth produces a density plot that has spikes as it captures the fine-scale variability in the distribution of values. As the bandwidth increases, the spikes disappear and the density gets smoother. The largest bandwidth produces a smooth symmetric density centered on the value of zero.

To create a density plot for the NAO values with a histogram overlay, type

```
> d = density(NAO$Jun, bw=.5)
> plot(d, main="", xlab="June NAO [s.d.]",
+      lwd=2, col="red")
> hist(NAO$Jun, prob=TRUE, add=TRUE)
> rug(NAO$Jun)
```

The density function takes your vector of data values as input and allows you to specify a bandwidth using the `bw` argument. Here you are using the vector of June NAO values and a bandwidth of .5 s.d. The bandwidth units are the same as the units of your data, here s.d. for the NAO. The output is saved as a density object, here called `d`. The object is then plotted using the `plot` method. You turn off the default plot title with the `main=""` and you specify a label for the values to be plotted below the horizontal axis. You specify the line width as 2 and the line color as red.

You then overlay the histogram using the `hist` function (see Figure 5.6). You use the `prob=TRUE` argument to make the bar height proportional to the density. The `add=TRUE` argument is needed so that the histogram plots on the same graph. One reason for plotting the histogram or density is to see whether your data can be described by a normal distribution. The Q-Q plot provides another way to make this assessment.

#### 5.1.4 Q-Q Plot

A Q-Q plot is a way to compare distributions. It does this by plotting quantile (Q) values of one distribution against the corresponding quantile (Q) values of the other

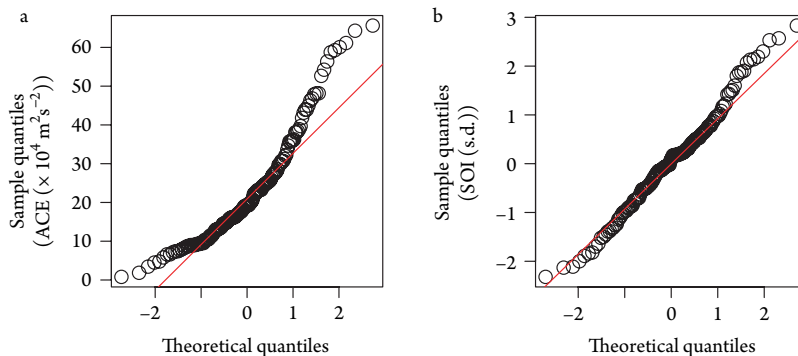


Figure 5.7 Q-Q normal plot of (a) ACE and (b) July SOI.

distribution. In the case of assessing whether or not your data are normally distributed, the sample quantiles are plotted on the vertical axis and quantiles from a standard normal distribution are plotted along the horizontal axis. In this case, it is called a Q-Q normal plot.

That is, the  $k$ th smallest observation is plotted against the expected value of the  $k$ th smallest random value from an  $N(0, 1)$  sample of size  $n$ . The pattern of points in the plot is then used to compare your data against a normal distribution. If your data are normally distributed then the points align along the  $y = x$  line shown on the plot.

This is done using the `qqnorm` function. To make side-by-side Q-Q normal plots for the ACE values and the July SOI values, you type

```
> par(mfrow=c(1, 2), pty="s")
> qqnorm(A$ACE)
> qqline(A$ACE, col="red")
> qqnorm(SOI$Jul)
> qqline(SOI$Jul, col="red")
```

The plots are shown in Figure 5.7. The quantiles are nondecreasing. The  $y = x$  line is added to the plot using the `qqline` function. Additionally, we adjusted the vertical axis label and turned the default title off.

The plots show that July SOI values appear to have a normal distribution while the seasonal ACE does not. For observations that have a positive skew, like the ACE, the pattern of points on a Q-Q normal plot is concave upward. For observations that have a negative skew, the pattern of points is concave downward. For values that have a symmetric distribution but with fatter tails than the normal (e.g., the  $t$ -distribution), the pattern of points resembles an inverse sine function.

The Q-Q normal plot is useful in checking the residuals from a regression model. The assumption is that the residuals are independent and identically distributed characterized by a normal distribution centered on zero. In Chapter 3, you created a multiple linear regression model for August SST using March SST and year as



explanatory variables. To examine the assumption of normally distributed residuals with a Q-Q normal plot, type

```
> model = lm(Aug ~ Year + Mar, data=SST)
> qqnorm(model$residuals)
> qqline(model$residuals, col="red")
```

Points align along the  $y = x$  axis indicating a normal distribution.

### 5.1.5 Scatter Plot

The `plot` function (method) is used to create a scatter plot. The values of one variable are plotted against the values of the other variable as points in a Cartesian plane (see Chapter 2). The values named in the first argument are plotted along the horizontal axis.

This pairing is useful in generating and testing hypotheses about a relationship between the two variables. In the context of correlation, which variable gets plotted on which axis is not of concern. Either way, the scatter of points illustrates the amount of correlation. However, in the context of a statistical model, by convention, the dependent variable (the variable you are interested in explaining) is plotted on the vertical axis and the explanatory variable is plotted on the horizontal axis. For example, if your interest is whether ACE is related to pre-hurricane season ocean warmth (e.g., June SST), your model is

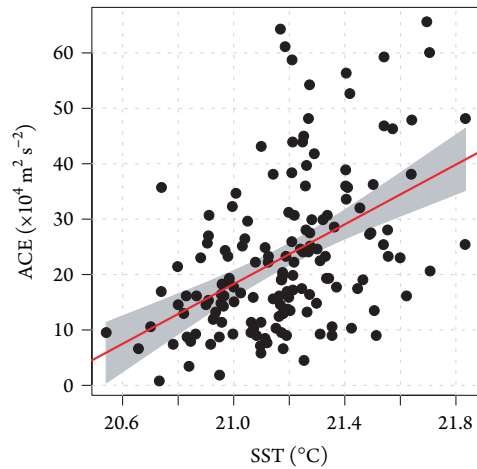
```
> ace = A$ACE*.5144^2
> sst = SST$Jun
> model = lm(ace ~ sst)
```

and you plot ACE on the vertical axis. Since your slope and intercept coefficients from the linear regression model are saved as part of the object `model`, you can first create a scatter plot and then use the `abline` function to add the linear regression line. Here the function extracts the intercept and slope coefficient values from the model object and draws the straight line using the point-intercept formula.

Here you use the model formula syntax (`ace ~ sst`) as the first argument in the `plot` function.


```
> plot(ace ~ sst, ylab=expression(
+   paste("ACE [x", 10^4, " ", m^2, s^-2, "]")),
+   xlab=expression(paste("SST [", degree, "C]")))
> abline(model, col="red", lwd=2)
```

Figure 5.8 is the result. The relationship between ACE and SST is summarized by the linear regression model shown by the straight line. The slope of the line indicates that for every  $1^{\circ}\text{C}$  increase in SST, the average value of ACE increases by  $27 \times 10^4 \text{ m}^2/\text{s}^2$  (type `coef(model[2])`).



**Figure 5.8** Scatter plot and linear regression line of ACE and June SST.

Since the regression line is based on a sample of data, you should display it inside a band of uncertainty. As we saw in Chapter 3, there are two types of uncertainty: a confidence band (narrow) and a prediction band (wide). The confidence band reflects the uncertainty about the line itself, which like the standard error of the mean indicates the precision by which you know the mean. Here the mean is not constant but rather a function of the explanatory variable.

The 95 percent confidence band is shown in Figure 5.8. The width of the band is inversely related to the sample size. In a large sample of  the confidence band will be narrow reflecting a well-determined line. Note that it is in this case impossible to draw a horizontal line that fits completely within the band. This indicates that there is a significant relationship between ACE and SST.

The band is narrowest in the middle, which is understood by the fact that the predicted value at the mean SST will be the mean of ACE, whatever the slope, and thus the standard error of the predicted value at this point is the standard error of the mean of ACE. At other values of SST, the variability associated with the estimated slope is included. This variability is larger for values of SST farther from the mean, which is why the band looks like a bow tie.

The prediction band adds another layer of uncertainty, the uncertainty about *future* values of ACE. The prediction band captures the majority of the observed points in the scatter plot. Unlike the confidence band, the width of the prediction band depends on the assumption of normally distributed errors with a constant variance across the values of the explanatory variable.

### 5.1.6 Conditional Scatter Plot

Scatter plots *conditional* on the values of a third variable can be quite informative. This is done with the `coplot` function. The syntax is the same as above except you add the name of the conditioning variable after a vertical bar.

For example, as SST increases so does ACE. The conditioning plot answers the question: is there a change in the relationship depending on values of the conditioning variable? Here you use August SOI values as the conditioning variable and type

```
> soi = SOI$Aug
> coplot(ace ~ sst | soi, panel=panel.smooth)
```

The syntax is read “conditioning plot of ACE versus SST given values of SOI.” The function divides the range of the conditioning variable (SOI) into six intervals (by default) with each interval having approximately the same number of years. The range of SOI values in each interval overlaps by 50 percent. The conditioning intervals are plotted in the top panel as horizontal bars (shingles). The plot is shown in Figure 5.9.

The scatter plots of ACE and SST are arranged in a matrix of panels below the shingles. The panels are arranged from lower left to upper right. The lower left panel corresponds to the lowest range of SOI values (less than about  $-1$  s.d.) and the upper right panel corresponds to the highest range of SOI values (greater than about  $+1.5$  s.d.). Half of the data points in a panel are shared with the panel to the left and half of the data points are shared with the panel to the right. This is indicated by the amount of shingle overlap.

Results show a positive, nearly linear, relationship between ACE and SST for all ranges of SOI values. Over the SOI range between  $-1.5$  and  $0$ , the relationship is somewhat curved. ACE is least sensitive to SST when SOI is the most negative (El Niño years) as indicated by the nearly flat line in the lower left panel. The argument `panel` adds a local linear curve (red line) through the set of points in each plot.

## 5.2 TIME SERIES

Hurricane data often take the form of a time series. A time series is a sequence of data values measured at successive times and spaced at uniform intervals. You can treat a time series as a vector and use structured data functions (see Chapter 2) to generate time series.

However, additional functions are available for data that are converted to time-series objects. Time series objects are created using the `ts` function. You do this with the monthly NAO data frame as follows. First create a matrix of the monthly values, skipping the year column in the data frame. Second take the transpose of this matrix (switch the rows with the columns) using the `t` function and save the matrix as a vector. Finally, create a time series object, specifying the frequency of values and the start month. Here the first value is from January 1851.

```
> nao.m = as.matrix(NAO[, 2:13])
> nao.v = as.vector(t(nao.m))
> nao.ts = ts(nao.v, frequency=12, start=c(1851, 1))
```

Also create a time series object for the cumulative sum of the monthly SOI values. This is done with the `cumsum` function applied to your data vector.

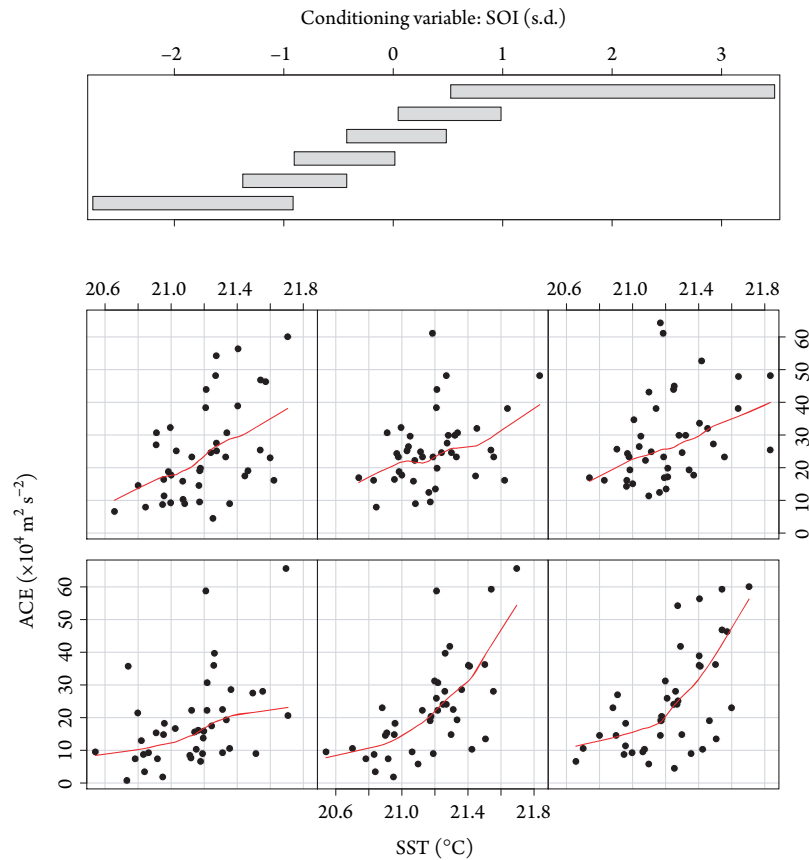


Figure 5.9 Scatter plots of ACE and SST conditional on the SOI.

```
> nao.cts = ts(cumsum(nao.v) ,
+ frequency=12, start=c(1851, 1))
```

This results in objects of class `ts`, which is used for time series having numeric time information. Additional classes for working with time series data that can handle dates and other types of time information are available. For example, the `fts` package implements regular and irregular time series based on `POSIXct` time stamps (see §5.2.3), and the `zoo` package provides functions for most time series classes.

### 5.2.1 Time Series Graph

The objects of class `ts` make it easy to plot your data as a time series. For instance, you plot the cumulative sum of the NAO values using the `plot` method. The method recognizes the object as a time series and plots it accordingly eliminating the need to specify a separate time variable.

```
> plot(nao.cts)
```

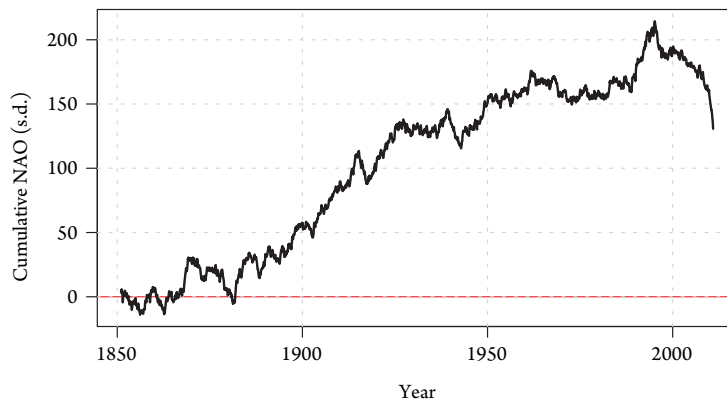


Figure 5.10 Time series of the cumulative sum of NAO values.

Figure 5.10 shows the result. The cumulative sum indicates a pattern typical of a random walk. That is, over the long term there is a tendency for more positive-value months leading to a “wandering” of the cumulative sum away from the zero line. This tendency begins to reverse in the late twentieth century.

### 5.2.2 Autocorrelation

Autocorrelation is correlation between values of a single variable. For time data it, refers to single series correlated with itself as a function of temporal lag. For spatial data, it refers to single variable correlated with itself as a function of spatial lag, which can be a vector of distance and orientation (see Chapter 9). In both cases, the term “autocorrelation function” is used, but with spatial data, the term is often qualified with the word “spatial.”

As an example, save 30 random values from a standard normal distribution in a vector where the elements are considered ordered in time. First, create a time series object. Then use the `lag.plot` function to create a scatter plot of the time series against a lagged copy where the lagged copy starts one time interval earlier.

```
> t0 = ts(rnorm(30))
> lag.plot(t0, lag=1)
```

With  $n$  values, the plot for lag one contains  $n - 1$  points. The points are plotted using the text number indicating the temporal order so that the first point labeled “1” is given by the coordinates  $(t0[1], t0[2])$ . The correlation at lag one can be inferred by the scatter of points. The plot can be repeated for any number of lags, but with higher lags, the number of points decreases.

You use the autocorrelation function (`acf`) to quantify the correlation at various temporal lags. The function accepts univariate and multivariate numeric time series objects and produces a plot of the autocorrelation values as a function of lag. For example, to create a plot of the autocorrelation function for the NAO time series object from the previous section, type

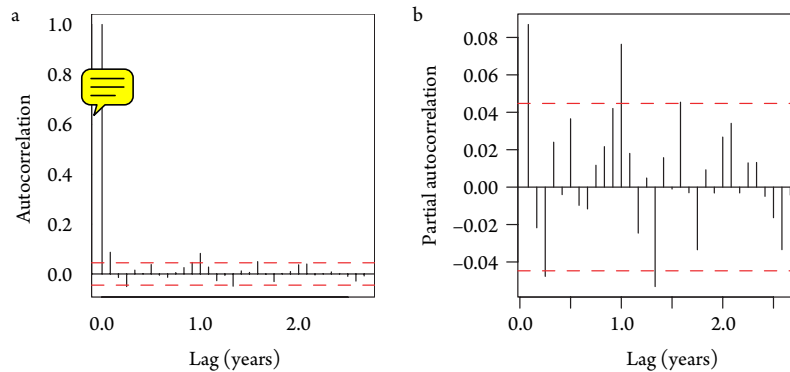


Figure 5.11 Autocorrelation and partial autocorrelation functions of monthly NAO.

```
> acf(nao.ts, xlab="Lag [Years]",
+     ylab="Autocorrelation")
```

The lag values on the horizontal axis are plotted in units of time rather than numbers of observations (see Fig. 5.11). Dashed lines are the 95 percent confidence limits. Here the time series object uses monthly frequency, so the lags are given in fractions of 12 with 1.0 corresponding to a year. The maximum lag is calculated as  $10 \times \log_{10} n$ , where  $n$  is the number of observations. This can be changed using the argument `lag.max`.

The lag-zero autocorrelation is fixed at 1 by convention. The nonzero autocorrelations are all less than 0.1 in absolute value indicative of an uncorrelated process. By default, the plot includes 95 percent confidence limits computed as  $\pm 1.96/\sqrt{n}$ .

The partial autocorrelation function `pacf` computes the autocorrelation at lag  $k$  after the linear dependencies between lags 1 to  $k - 1$  are removed. The partial autocorrelation is used to identify the temporal extent of the autocorrelation. Here the partial autocorrelation vacillates between positive and negative values indicative of a moving-average process.<sup>1</sup>

If your regression model uses time series data, it is important to examine the autocorrelation in the model residuals. If residuals from your regression model have significant positive autocorrelation, then the assumption of independence is violated. This violation does not bias the coefficient estimates, but standard errors on the coefficients tend to be too small giving you too much confidence in your inferences.

### 5.2.3 Dates and Times

Various options exist for working with date and time data in R. The `as.Date` function gives you flexibility in handling dates through the `format` argument. The default

<sup>1</sup> A moving-average process is one in which the expectation of the current value of the series is linearly related to previous white noise (uncorrelated) errors.

**Table 5.1** Format codes for dates.

<i>Code</i>	<i>Value</i>
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated, e.g., Jan)
%B	Month (full name)
%y	Year (2 digit)
%Y	Year (4 digit)

format is a four-digit year, a month, then a day, separated by dashes or slashes. For example, the character string "1992-8-24" will be accepted as a date by typing

```
> Andrew = as.Date("1992-8-24")
```

Although the print method displays it as a character string, the object is a `Date` class stored as the number of days since January 1, 1970, with negative numbers for earlier dates.

If your input dates are not in the standard year-month-day order, a format string can be composed using the elements shown in Table 5.1. For instance, if your date is specified as August 29, 2005, then you type

```
> Katrina = as.Date("August 29, 2005",
+   format="%B %d, %Y")
```

You can find the number of days between hurricanes Andrew and Katrina by typing

```
> difftime(Katrina, Andrew, units="days")
Time difference of 4753 days
```

Or you can obtain the number of days from today since Andrew by typing

```
> difftime(Sys.Date(), Andrew, units="days")
```

The function `Sys.Date` with no arguments gives the current day in year-month-day format as a `Date` object.

The portable operating system interface (POSIX) has formats for dates and times, with functionality for converting between time zones (Spector, 2008). The POSIX date/time classes store times to the nearest second. There are two such classes differing only in the way the values are kept internally. The `POSIXct` class stores date/time values as the number of seconds since January 1, 1970, while the `POSIXlt` class stores them as a list. The list contains elements for second, minute, hour, day, month, and year among others.

The default input format for POSIX dates consists of the year, month, and day, separated by slashes or dashes with time information followed after a space. The

time information is in the format `hour:minutes:seconds` or simply `hour:minutes`. For example, according to the U.S. National Hurricane Center, Hurricane Andrew hit Homestead Air Force Base at 0905 UTC on August 24, 1992. You add time information to your Andrew date object and convert it to a `POSIXct` object by typing.

```
> Andrew = as.POSIXct(paste(Andrew, "09:05"),
+   tz="GMT")
```

You then retrieve your local time from your operating system as a character string and use the date–time conversion `strptime` function to convert the string to a `POSIXlt` class.

```
> mytime = strptime(Sys.time(), format=
+   "%Y-%m-%d %H:%M:%S", tz="EST5EDT")
```

Our time zone is U.S. Eastern standard time, so we use `tz="EST5EDT"`. You then find the number of hours since Andrew’s landfall by typing,

```
> difftime(mytime, Andrew, units="hours")
Time difference of 171482 hours
```

Note that time zones are not portable, but `EST5EDT` comes pretty close.

Additional functionality for working with times is available in the **chron** and **lubridate** packages. In particular, **lubridate** (great package name) makes it easy to work with dates and times by providing functions to identify and parse date–time data, extract and modify components (years, months, days, hours, minutes, and seconds), perform math on date–times, and handle time zones and Daylight Savings Time (Grolemund and Wickham, 2011).

For example, to return the day of the week from your object `Andrew`, you use the `wday` function in the package by typing

```
> require(lubridate)
> wday(Andrew, label=TRUE, abbr=FALSE)
[1] Monday
7 Levels: Sunday < Monday < ... < Saturday
```

If you lived in south Florida, what a Monday it was. Other examples of useful functions in the package related to the Andrew time object include, the year, was it a leap year, what week of the year was it, and what local time was it. Finally, what is the current time in Chicago?

```
> year(Andrew)
> leap_year(Andrew)
> week(Andrew)
> with_tz(Andrew, tz="America/New_york")
> now(tz="America/Chicago")
```



### 5.3 MAPS

A great pleasure in working with graphs is the chance to visualize patterns. Maps are among the most compelling graphs as the space they map is the space in which hurricanes occur. We can use them to find interesting, sometimes hidden, patterns. Various packages are available for creating maps. Here we look at a few examples.

#### 5.3.1 Boundaries

Sometimes all you need is a reference map to show your study location. This can be created using state and country boundaries. For example, the **maps** package is used to draw country and state borders. To draw a map of the United States with state boundaries, type

```
> require(maps)
> map("state")
```

The call to `map` creates the country outline and adds the state boundaries. The map is shown in Figure 5.12. The package contains outlines for countries around the world (e.g., type `map()`).

The coordinate system is latitude and longitude, so you can overlay other spatial data. As an example, first input the track of Hurricane Ivan (2004) as it approached the U.S. Gulf coast. Then list the first six rows of data.

```
> Ivan = read.table("Ivan.txt", header=TRUE)
> head(Ivan)
  Year Mo Da Hr  Lon  Lat Wind WindS Pmin Rmw  Hb
1 2004  9 15  8 -87.6 25.9 118  118  917 37 1.27
2 2004  9 15  9 -87.7 26.1 118  117  917 37 1.27
3 2004  9 15 10 -87.7 26.3 117  116  917 37 1.26
4 2004  9 15 11 -87.8 26.5 117  116  918 37 1.26
```

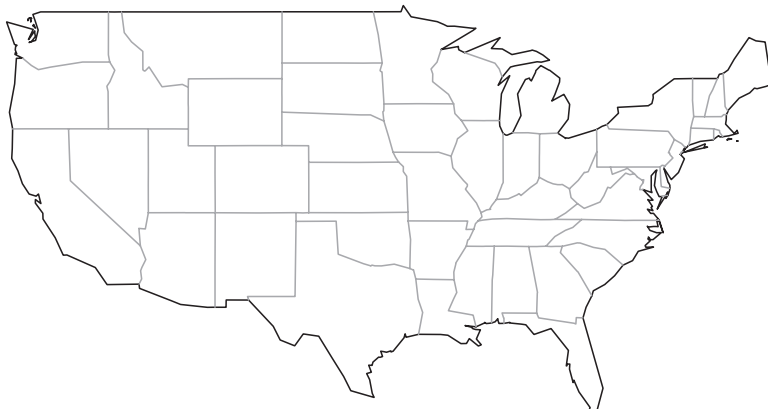


Figure 5.12 Map with state boundaries.

```

5 2004  9 15 12 -87.9 26.7 117 115 918 37 1.26
6 2004  9 15 13 -88.0 26.9 116 115 919 37 1.26
  Speed L Lhr
1  11.6 0 -24
2  12.1 0 -23
3  12.4 0 -22
4  12.6 0 -21
5  12.7 0 -20
6  12.8 0 -19

```

Among other attributes, the data frame `Ivan` contains the latitude and longitude position of the hurricane center every hour from 24 hours before landfall until 12 hours after landfall.

Here your geographic domain is the southeast, so first create a character vector of state names.

```

> cs = c('texas', 'louisiana', 'mississippi',
+       'alabama', 'florida', 'georgia', 'south carolina')

```

Next use the `map` function with this list to plot the state boundaries and fill the state polygons with a gray shade. Finally, connect the hourly location points with the `lines` function and add an arrowhead to the last two locations.

```

> map("state", region=cs, boundary=FALSE, col="gray",
+     fill=TRUE)
> Lo = Ivan$Lon
> La = Ivan$Lat
> n = length(Lo)
> lines(Lo, La, lwd=2.5, col="red")
> arrows(Lo[n - 1], La[n - 1], Lo[n], La[n], lwd=2.5,
+       length=.1, col="red")

```

The result is shown in Figure 5.13. Hurricane Ivan moved northward from the central Gulf of Mexico and makes landfall in the western panhandle region of Florida before moving into southeastern Alabama.

The scale of the map is defined as the ratio of the map distance in a particular unit (e.g., centimeters) to the actual distance in the same unit. Small scale describes maps of large regions where this ratio is small and large scale describes maps of small regions where the ratio is larger. The boundary data in the `maps` package is sufficient for use with small-scale maps but the number of boundary points is not sufficient for large-scale maps (close-up or high resolution). Higher-resolution boundary data are available in the `mapdata` package.

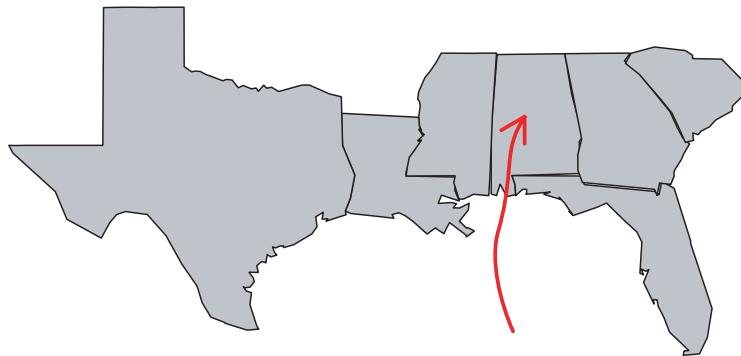


Figure 5.13 Track of Hurricane Ivan (2004) before and after landfall.

### 5.3.2 Data Types

The type of map you make will depend on the type of spatial data you have. Broadly speaking, there are three types of data: point, areal, and field data. Point data are event locations. Any location in a continuous spatial domain may have an event. The events may carry additional information, called “marks.” Interest centers on the distribution of events and on whether there are event clusters. The set of all locations where hurricanes first reached maximum intensity is an example of point data. The events are the location of the hurricane at maximum intensity, and a mark is the corresponding wind speed.

Areal data are values aggregated within fixed polygons. The set of polygons form a lattice so areal data are called “lattice data.” Interest centers on how the values change across the domain and on how much correlation exists within neighborhoods defined by contiguity or distance. County-wide population is an example of areal data. The values may be the number of people living in the county or a population density indicating the average number of people per area.

Field data are observations of a spatially continuous variable, like pressure or temperature. The values are given at certain locations and the interest centers on using these values to create a continuous surface from which inferences can be made at any location. Sea-level pressure is an example of field data.

#### Point Data

Consider the set of events defined by the location at which a hurricane first reaches lifetime maximum intensity. The data are available in the file *LMI.txt* and are input by typing

```
> LMI.df = read.table("LMI.txt", header=TRUE)
> LMI.df$WmaxS = LMI.df$WmaxS * .5144
> head(LMI.df[, c(4:10, 11)])
      name  Yr Mo Da hr  lon  lat  Wmax
30861.5 DENNIS    1981  8 20 23 -70.8 37.0 70.4
```

30891.4	EMILY	1981	9	6	10	-58.1	40.6	80.6
30930.2	FLOYD	1981	9	7	2	-69.1	26.8	100.4
30972.2	GERT	1981	9	11	14	-71.7	29.4	90.5
31003.5	HARVEY	1981	9	14	23	-62.6	28.3	115.1
31054.4	IRENE	1981	9	28	16	-56.4	27.9	105.5

The `wmax` column is a spline-interpolated maximum wind speed and `wmaxS` (not shown) is first smoothed then spline interpolated to allow time derivatives to be computed. Chapter 6 provides more details and explains how this data set is constructed.

The raw wind speed values are given in 5-kt increments. Although knots (kt) are the operational unit used for reporting tropical cyclone intensity to the public in the United States, here you use the SI units of  $\text{m s}^{-1}$ . We use the term “intensity” as shorthand for “maximum wind speed,” where maximum wind speed refers to the estimated fastest wind velocity somewhere in the core of the hurricane. Lifetime maximum refers to the fastest wind during the life of the hurricane.

You draw a map of the event locations with the `plot` method using the longitude coordinate as the  $x$  variable and the latitude coordinate as the  $y$  variable by typing

```
> with(LMI.df, plot(lon, lat, pch=19))
> map("world", col="gray", add=TRUE)
> grid()
```

Adding country borders and latitude/longitude grid lines (`grid` function) enhances the geographic information. The argument `pch` specifies a point character using an integer code. Here 19 refers to a solid circle (type `?points` for more information). The `with` function allows you use the column names from the data frame in the `plot` method.

Note the order of function calls. By plotting the events first, then adding the country borders, the borders are clipped to the plot window. The dimensions of the plot window are slightly larger than the range of the longitude and latitude coordinates. The function chooses a reasonable number of axis ticks that are placed along the range of coordinate values at reasonable intervals.

Since the events are marked by storm intensity, it is informative to add this information to the map. Hurricane intensity, as indexed by an estimate of the wind speed maximum, is a continuous variable. You can choose a set of discrete intensity intervals and group the events by these class intervals. For example, you might want to choose the Saffir–Simpson hurricane intensity scale.

To efficiently communicate differences in intensities with colors, you should limit the number classes to six or less. The package `classInt` is a collection of functions for choosing class intervals. Here you require the package and create a vector of lifetime maxima. You then obtain class boundaries using the `classIntervals` function. Here the number of class intervals is set to five and the method of determining the interval breaks is based on Jenks optimization (`style="jenks"`). Given the number of classes, the optimization minimizes the variance of the values within the intervals while maximizing the variance between the intervals.

```
> require(classInt)
> lmi = LMI.df$WmaxS
> q5 = classIntervals(lmi, n=5, style="jenks",
+   dataPrecision=1)
```

The `dataPrecision` argument sets the number of digits to the right of the decimal place.

Next you choose a palette of colors. This is best left to someone with an understanding of hues and color separation schemes. The palettes described and printed in Brewer et al. (2003) for continuous, diverging, and categorical variables can be examined on maps at <http://colorbrewer2.org/>. Select the HEX radio button for a color palette of your choice and then copy and paste the hex code into a character vector preceded by the pound symbol.

For example, here you create a character vector (`cls`) of length 5 containing the hex codes from the color brewer website from a sequential color ramp ranging among yellow, orange, and red.

```
> cls = c("#FFFFB2", "#FECC5C", "#FD8D3C", "#F03B20",
+   "#BD0026")
```

To use your own set of colors, simply modify this list. A character vector of color hex codes is generated automatically with functions in the **colorRamps** package (see Chapter 9).

The empirical cumulative distribution function of cyclone intensities with the corresponding class intervals and colors is plotted by typing

```
> plot(q5, pal=cls, main="", xlab=
+   expression(paste("Wind Speed [m ", s^-1, "]")),
+   ylab="Cumulative Frequency")
```

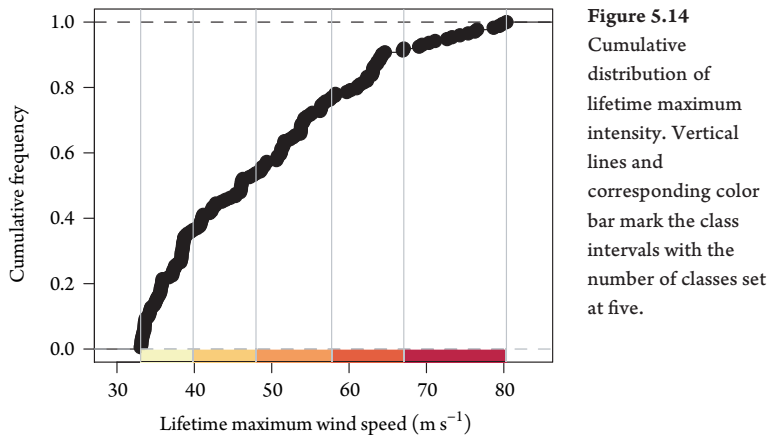
The graph is shown in Figure 5.14. The points (with horizontal dashes) are the lifetime maximum intensities in rank order from lowest to highest. You can see that half of all hurricanes have lifetime intensities greater than  $46 \text{ m s}^{-1}$ .

Once you are satisfied with the class intervals and color palette, you can plot the events on a map. First you need to assign a color for each event depending on its wind speed value. This is done with the `findColours` function as

```
> q5c = findColours(q5, cls)
```

Now, instead of black dots with a color bar, each value is assigned a color corresponding to the class interval. For convenience, you create the axis labels and save them as expression objects. You do this with the `expression` and `paste` functions to get the degree symbol.

```
> xl = expression(paste("Longitude [", {}^o, "E"]))
> yl = expression(paste("Latitude [", {}^o, "N"]"))
```



**Figure 5.14**  
Cumulative distribution of lifetime maximum intensity. Vertical lines and corresponding color bar mark the class intervals with the number of classes set at five.

Since the degree symbol is not attached to a character, you use `{ }` in front of the superscript symbol. You again use the `plot` method on the location coordinates, but this time set the color argument to the corresponding vector of colors saved in `q5c`.

```
> plot(LMI.df$lon, LMI.df$lat, xlab=x1, ylab=y1,
+      col=q5c, pch=19)
> points(LMI.df$lon, LMI.df$lat)
```

To improve the map, you add country boundaries, place axis labels in the top and right margins, and add a coordinate grid.

```
> map("world", add=TRUE)
> axis(3)
> axis(4)
> grid()
```

To complete the map, you add a legend by typing

```
> legend("bottomright", bg="white",
+       fill=attr(q5c, "palette"),
+       legend=names(attr(q5c, "table")),
+       title=expression(paste("Wind Speed [m ",
+                               , s^-1, "]")))
```

Note that fill colors and names for the legend are obtained using the `attr` function on the `q5c` object. The function retrieves the table attribute of the object. The result is shown in Figure 5.15. Colors indicate the wind speed in five classes as described in Figure 5.14.

The spatial distribution of lifetime maxima is fairly uniform over the ocean for locations west of the  $-40^{\circ}\text{E}$  longitude. Fewer events are noted over the eastern Caribbean Sea and southwestern Gulf of Mexico. Events over the western Caribbean tend to

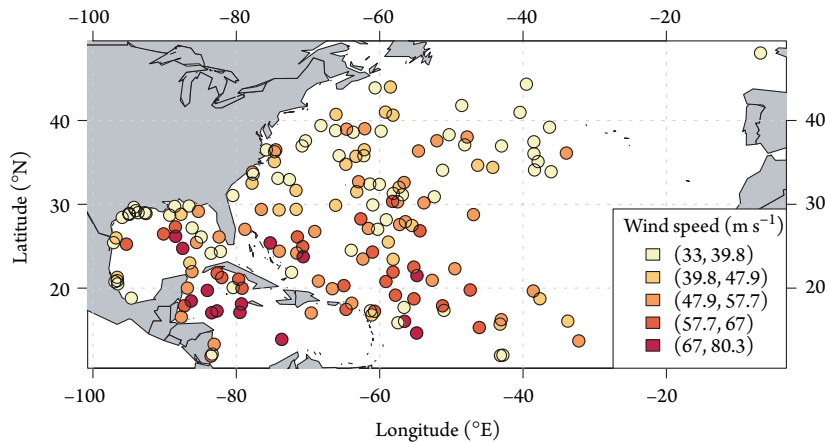


Figure 5.15 Location of lifetime maximum wind speed.

have the highest intensities. Also there is a tendency for hurricanes reaching lifetime maximum at lower latitudes to have higher intensities.

### Areal Data

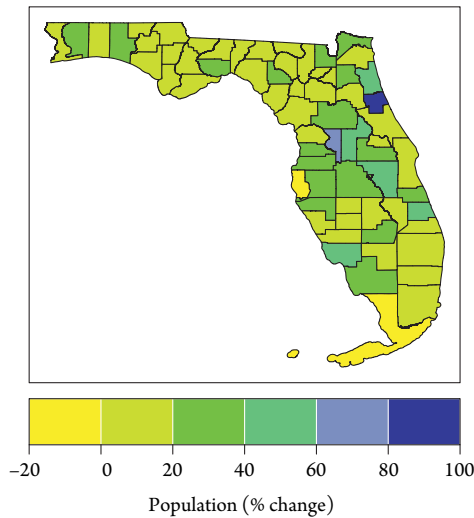
A shapefile stores geometry and attribute information for spatial data. The geometry is a set of vector coordinates. Shapefiles support point, line, and area data. Area data are represented as closed-loop polygons. Each attribute record has a one-to-one relationship with the associated shape record. For example, a shapefile might consist of the set of polygons for the counties in Florida and an attribute might be population. Associated with each county, population record (attribute) is an associated shape record.

The shapefile is actually a set of several files in a directory. The three files with extensions `*.shp` (file of geometries), `*.shx` (index file to the geometries), and `*.dbf` (file for storing attribute data) form the core of the directory. Note that there is no standard for specifying missing attribute values. The `*.prj` file, if present, contains the coordinate reference system (CRS; see §5.4).

Information in a shapefile format makes it easy to map. As an example, consider the U.S. Census Bureau boundary file for the state of Florida <http://www.census.gov/cgi-bin/geo/shapefiles/national-files>. Browse to Current State and Equivalent, Select State, then Florida. Download the zipped file. Unzip it to your R working directory. To make things a bit easier for typing, rename the directory and the shapefiles to `FL`.

The `readShapeSpatial` function from the **maptools** package reads in the polygon shapefile consisting of the boundaries of the 67 Florida counties.

```
> require(maptools)
> FLpoly = readShapeSpatial("FL/FL")
> class(FLpoly)
```



**Figure 5.16**  
Population change in  
Florida counties

```
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

Note the shapefiles are in directory FL with file names the same as the directory name. The object `FLpoly` created is a `SpatialPolygonsDataFrame` class that extends the class `data.frame` by adding geographic information (see Bivand et al. (2008)).

You can use the `plot` method to produce a map of the polygon borders. More interestingly is a map displaying an attribute of the polygons. For instance, demographic data at the county level are important for emergency managers. First read in a table of the percentage change in population over the 10-year period 2000–2010.

```
> FLPop = read.table("FLPop.txt", header=TRUE)
> names(FLPop)
[1] "Order"   "County"  "Pop2010" "Pop2000" "Diff"
[6] "Change"
```

Here the table rows are arranged in the same order as the polygons. You assign the column `Change` to the data slot of the spatial data frame by typing

```
> FLpoly$Change = FLPop$Change
```

Then use the function `splot` to create a choropleth map of the attribute `Change`.

```
> splot(FLpoly, "Change")
```

Results are shown in Figure 5.16. With the exception of Monroe and Pinellas counties, population throughout the state increased over this period. Largest population increases are noted over portions of northern Florida.



The `spplot` method is available in the `sp` package. It is an example of a lattice plot method (Sarkar, 2008) for spatial data with attributes. The function returns a plot of class `trellis`. If the function does not automatically bring up your graphics device, you need to wrap it in the `print` function. Missing values in the attributes are not allowed.

### Field Data

Climate data are often presented as values on a grid. For example, NOAA-CIRES 20th Century Reanalysis version 2 provides monthly sea-surface temperatures at latitude–longitude intersections. A portion of these data are available in the file *JulySST2005.txt*. The data are the SST values on a  $2^\circ$  latitude–longitude grid for the month of July 2005. The grid is bounded by  $-100^\circ$  and  $10^\circ$ E longitudes and the equator and  $70^\circ$ N latitude.

First input the data and convert the column of SST values to a matrix using the `matrix` function specifying the number of columns as the number of longitudes. The number of rows is inferred based on the length of the vector. Next create two structured vectors, one of the meridians and the other of the parallels using the `seq` function. Specify the geographic limits and an interval of  $2^\circ$  in both directions.

```
> sst.df = read.table("JulySST2005.txt", header=TRUE)
> sst = matrix(sst.df$SST, ncol=36)
> lo = seq(-100, 10, 2)
> la = seq(0, 70, 2)
```

To create a map of the SST field, first choose a set of colors. Since the values represent temperature, you want the colors to go from blue (cool) to red (warm). R provides a number of color palettes including `rainbow`, `heat.colors`, `cm.colors`, `topo.colors`, `grey.colors`, and `terrain.colors`. The palettes are functions that generate a sequence of color codes interpolated between two or more colors. The `cm.colors` is the default palette in `sp.plot` and the colors diverge from white to cyan and magenta.

More color options from the Web site are given in §5.3.2. The package **RColorBrewer** provides the palettes described in Brewer et al. (2003). Palettes are available for continuous, diverging, and categorical variables and for choices of print and screen projection. The `sp` package has the `bpy.colors` function that produces a range of colors from blue to yellow that work for color and black-and-white print. You can create your own palette using the `colorRampPalette` function. Here you save the function as `bwr` and use a set of three colors. The number of colors to interpolate is the argument to the `bwr` function.

```
> bwr = colorRampPalette(c("blue", "white", "red"))
```

The `image` function creates a grid of rectangles with colors corresponding to the values in the third argument as specified by the palette and the number of colors set here at 20. The first two arguments correspond to the two-dimensional location of the

rectangles. The  $x$  and  $y$  labels use the `expression` and `paste` functions to get the degree symbol. You add country boundaries and large axis labels in the top and right margins (margins 3 and 4) to complete the graph.

```
> image(lo, la, sst, col=bwr(20), xlab=x1, ylab=y1)
> map("world", add=TRUE)
> axis(3)
> axis(4)
```

Note that `image` interprets the matrix of SST values as a table with the  $x$ -axis corresponding to the row number and the  $y$ -axis to the column number, with column one at the bottom. This is an orthogonal counterclockwise rotation of the conventional matrix layout.

Overlay a contour plot of the SST data using the `contour` function. First determine the range of the SST values and round to the nearest whole integer. There are missing values (over land) so you need to use the `na.rm` argument in the `range` function.

```
> r = round(range(sst, na.rm=TRUE))
```

Next create a string of temperature values at equal intervals within this range. Contours will be drawn at these values.

```
> levs = seq(r[1], r[2], 2)
> levs
[1] -2  0  2  4  6  8 10 12 14 16 18 20 22 24 26 28
[17] 30
```

Then paste the character string “C” onto the interval labels. The corresponding list is used as contour labels.

```
> cl = paste(levs, "C")
> contour(lo, la, sst, levels=levs, labels=cl,
+   add=TRUE)
```

The result is shown in Figure 5.17. Ocean temperatures above about 28°C are warm enough to support the development of hurricanes. This covers a large area from the west coast of Africa westward through the Caribbean and Gulf of Mexico and northward toward Bermuda.

## 5.4 COORDINATE REFERENCE SYSTEMS

For data covering a large geographic area, you need a map with a projected coordinate reference system (CRS). A geographic CRS includes a model for the shape of the earth (oblate spheroid) plus latitudes and longitudes. Longitudes and latitudes can be used to create a two-dimensional coordinate system for plotting hurricane data, but this framework is for a sphere and not a flat map.

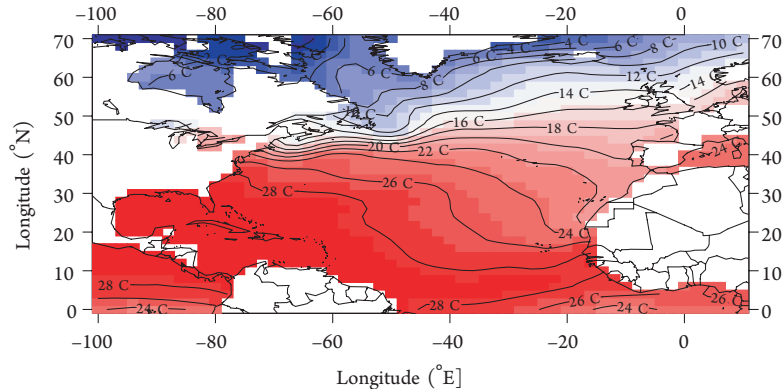


Figure 5.17 Sea-surface temperature field from July 2005.

A projected CRS is a two-dimensional approximation of the earth as a flat surface. It includes a model for the earth's shape plus a specific geometric model for projecting coordinates onto the plane. The PROJ.4 Cartographic Projections library uses a `tab = value` representation of a CRS, with a tag and value pair within a single character string. The Geospatial Data Abstraction Library (GDAL) contains code for translating between different CRSs. Both the PROJ.4 and GDAL libraries are available in the `rgdal` package (Keitt et al., 2012). Here you specify a geographic CRS and save it in a CRS object called `ll_crs` (lat-lon coordinate reference system).

```
> require(rgdal)
> require(mapdata)
> ll_crs = CRS("+proj=longlat +ellps=WGS84")
```

The only values used autonomously in CRS objects are whether the string is a character `NA` (missing) for an unknown CRS, and whether it contains the string `longlat`, in which case the CRS is geographic (Bivand et al., 2008).

There are a number of different tags, beginning with "+", and separated from the value with "=", using white space to separate the tag/value pairs. Here you specify the earth's shape using the World Geodetic System (WGS) 1984, which is the reference coordinate system used by the Global Positioning System to reference the earth's center of mass.

As an example, you create a `SpatialPoints` object called `LMI_ll` by combining the matrix of event coordinates (location of lifetime maximum intensity) in native longitude and latitude degrees with the CRS object defined above.

```
> LMI_mat = cbind(LMI.df$lon, LMI.df$lat)
> LMI_ll = SpatialPoints(LMI_mat,
+   proj4string=ll_crs)
> summary(LMI_ll)
Object of class SpatialPoints
Coordinates:
```

```

                min    max
coords.x1 -97.1 -6.87
coords.x2  11.9 48.05
Is projected: FALSE
proj4string : [+proj=longlat +ellps=WGS84]
Number of points: 173

```

Here you are interested in transforming the geographic CRS into a Lambert conformal conic (LCC) planar projection. The projection superimposes a cone over the earth, with two reference parallels secant to the globe. The LCC projection is used for aeronautical charts. It is used by the U.S. NHC in their seasonal summary maps. Other projections, ellipsoids, and datum are available, and a list of the various tag options can be generated by typing

```
> projInfo(type = "proj")
```

Besides the projection tag (`lcc`), you need to specify the two secant parallels and a meridian. The NHC summary maps use the parallels 30 and 60°N and a meridian of 60°W. First save the CRS as a character string, then use the `spTransform` function to transform the longitude–latitude coordinates to coordinates of a LCC planar projection.

```

> lcc_crs = CRS("+proj=lcc +lat_1=60 +lat_2=30
+ +lon_0=-60")
> LMI_lcc = spTransform(LMI_ll, lcc_crs)

```

This transforms the original set of longitude–latitude coordinates to a set of projected coordinates. You need to repeat this transformation for each of the map components. For instance, to transform the country borders, first save them from a call to the `map` function. The function includes arguments to specify a longitude–latitude bounding box. Second, convert the returned map object to a spatial lines object with the `map2SpatialLines` function using a geographic CRS. Finally, transform the coordinates of the spatial lines object to the LCC coordinates.

```

> brd = map('world', xlim=c(-100, 0), ylim=c(5, 50),
+ interior=FALSE, plot=FALSE)
> brd_ll = map2SpatialLines(brd, proj4string=ll_crs)
> brd_lcc = spTransform(brd_ll, lcc_crs)

```

To include longitude–latitude grid lines, you use the `gridlines` function on the longitude–latitude borders and then transform them to LCC coordinates. Similarly, to include grid labels, you convert the locations in longitude–latitude space to LCC space.

```

> grd_ll = gridlines(brd_ll)
> grd_lcc = spTransform(grd_ll, lcc_crs)
> at_ll = gridat(brd_ll)
> at_lcc = spTransform(at_ll, lcc_crs)

```

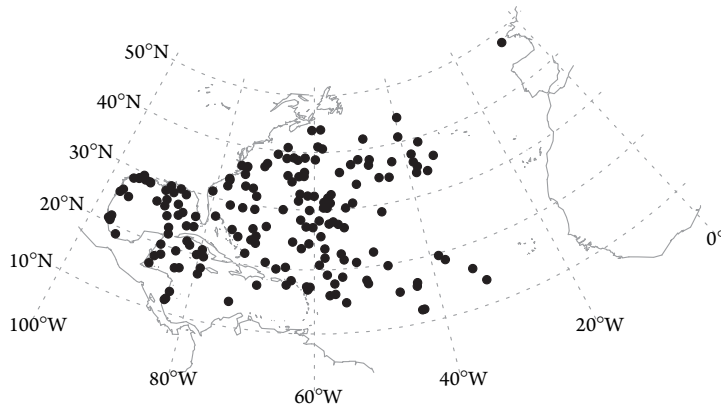


Figure 5.18 Lifetime maximum intensity events on a Lambert conic conformal map.

Finally, to plot the events on a projected map, first plot the grid and then add the country borders and event locations. Use the `text` function to add grid labels and include a box around the plot.

```
> plot(grd_lcc, col="grey60", lty="dotted")
> plot(brd_lcc, col="grey60", add=TRUE)
> plot(LMI_lcc, pch=19, add=TRUE, cex=.7)
> text(coordinates(at_lcc), pos=at_lcc$pos,
+       offset=at_lcc$offset-.3, labels=
+       parse(text=as.character(at_lcc$labels)),
+       cex=.6)
```

The result is shown in Figure 5.18. Conformal maps preserve angles and shapes of small figures, but not size. The size distortion is zero at the two reference latitudes. These features are useful for hurricane tracking maps.

The `splot` method for points, lines, and polygons has advantages over successive calls to `plot`. Chapter 9 contains examples.

## 5.5 EXPORT

The `rgdal` package has drivers for reading and writing spatial vector data using the OGR<sup>2</sup> Simple Features Library modeled on the OpenGIS simple features data model supported by the Open Geospatial Consortium, Inc.<sup>®</sup> If the data have a CRS, it will be read and written. The availability of OGR drivers depends on your computer. To get a list of the drivers available type `ogrDrivers()`.

<sup>2</sup> Historically, OGR was an abbreviation for “OpenGIS Simple Features Reference Implementation.” However, since OGR is not fully compliant with the OpenGIS Simple Feature specification and is not approved as a reference implementation, the name was changed to “OGR Simple Features Library.” OGR is the prefix used everywhere in the library source for class names, filenames, and so on.

Here you consider two examples. First export the lifetime maximum intensity events as a Keyhole Markup Language (KML) for overlay using Google Earth™ and then export the events as an ESRI™ shapefile suitable for input into ArcMap®, the main component of ESRI™'s Geographic Information System (GIS).

First create a spatial points data frame from the spatial points object. This is done using the `SpatialPointsDataFrame` function. The first argument is the coordinates of the spatial points object. The underlying CRS for Google Earth™ is geographical in the WGS84 datum, so you use the `LMI_11` object defined above and specify the argument `proj4string` as the character string `11_crs`, also defined earlier.

```
> LMI_sdf = SpatialPointsDataFrame(coordinates(LMI_11),
+   proj4string=11_crs, data=as(LMI.df, "data.frame")
+   [c("WmaxS")])
> class(LMI_sdf)
[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

The resulting spatial points data frame (`LMI_sdf`) contains a data slot with a single variable `WmaxS` from the `LMI.df` data frame, which was specified by the `data` argument.

To display the structure of the object, type

```
> str(LMI_sdf, max.level=3)
```

The argument `max.level` specifies the level of nesting (e.g., lists containing sub lists). By default, all nesting levels are shown, and this can produce too much output for spatial objects. Note that there are five slots with names `data`, `coordinates`, `coords`, `bbox`, and `proj4string`. The data slot is a regular data frame, here containing a single variable.

The `writeOGR` function takes as input the spatial data frame object and the name of the data layer and outputs a file in your working directory with a name given by the `dsn` argument and in a format given by the `driver` argument.

```
> writeOGR(LMI_sdf, layer="WmaxS", dsn="LMI.kml",
+   driver="KML", overwrite_layer=TRUE)
```

The resulting file can be viewed in Google Earth™ with pushpins for event locations. The pins can be selected revealing the layer values. You will see how to create an overlay image in Chapter 9.

You can also export to a shapefile. First transform your spatial data frame into the Lambert conformal conic used by the NHC.

```
> LMI_sdf2 = spTransform(LMI_sdf, lcc_crs)
> str(LMI_sdf2, max.level=2)
```

Note that the coordinate values are not longitude and latitude and neither are the dimensions of the bounding box (bbox slot).

You export using the driver ESRI Shapefile. The argument dsn is a folder name.

```
> drv = "ESRI Shapefile"
> writeOGR(LMI_sdf2, layer="WmaxS", dsn="WmaxS",
+ driver=drv, overwrite_layer=TRUE)
```

The output contains a set of four files in the WmaxS folder including a .prj file with the fully specified CRS. The data can be imported as a layer to ArcMap®.

## 5.6 OTHER GRAPHIC PACKAGES

R's traditional (standard) graphics offer a nice set of tools for making statistical plots including box plots, histograms, and scatter plots. The plots are produced using a single function. Yet some plots require a lot of work and even simple changes can be tedious. This is particularly true when you want to make a series of related plots for different partitions of your data. Two alternatives to the standard graphics are worth mentioning.

### 5.6.1 lattice

The **lattice** package (Sarkar, 2008) contains functions for creating trellis graphs for a variety of plot types. A trellis graph displays a variable or the relationship between variables, conditioned on another variable(s).

In simple usage, lattice functions work like traditional graphics functions. As an example of a lattice graphic function that produces a density plot of the June NAO values, type

```
> require(lattice)
> densityplot(~ Jun, data=NAO)
```

The function's syntax includes the name of the variable and the name of the data frame. The variable is preceded by the tilde symbol. By default, the density plot includes the values as points jittered above the horizontal axis.

The power of trellis graphs comes from being able to easily create a series of plots with the same axes (trellis) as you did with the `coplot` function in §5.1.6. For instance, in an exploratory analysis, you might want to see if the annual U.S. hurricane count is related to the NAO. You first create a variable that splits the NAO into four groups.

```
> steer = equal.count(NAO$Jun, number=4, overlap=.1)
```

The grouping variable has class `shingle` and the number of years in each group is the same. The `overlap` argument indicates the fraction of overlap in the data used

to group the years. If you want to leave gaps, you specify a negative fraction. You can type `plot(steer)` to see the range of values for each group.

Next you use the `histogram` function to plot the percentage of hurricanes by count conditional on your grouping variable.

```
> histogram(~ US$All | steer, breaks=seq(0, 8))
```

The vertical line indicates that the conditioning variable follows. The `breaks` argument is used on the hurricane counts. The resulting four-panel graph is arranged from lower left to upper right with increasing values of the grouping variable. Each panel contains a histogram of U.S. hurricane counts drawn using an identical scale for the corresponding range of NAO values. The relative range is shown above each panel in a strip (`shingle`).

Lattice functions produce an object of class `trellis` that contains a description of the plot. First assign it to the object `dplot` then print it. The print method for objects of this class does the actual drawing of the plot. For example, the following code does the same as shown previously.

```
> dplot = densityplot(~Jun, data=NAO)
> print(dplot)
```

Now you can use the `update` function to modify the plot design. For example, to add an axis label, type

```
> update(dplot, xlab="June NAO (s.d.)")
```

To save the modified plot for additional changes, you need to reassign it.

## 5.6.2 ggplot2

The **ggplot2** package (Wickham, 2009) contains plotting functions that are more flexible than the traditional R graphics. The `gg` stands for the “Grammar of Graphics,” a theory of how to create a graphics system (Wilkinson, 2005). The grammar specifies how a graphic maps data to attributes of geometric objects. The attributes are things like color, shape, and size, and the geometric objects are things like points, lines, bars, and polygons.

The plot is drawn on a specific coordinate system (which can be geographic) and it may contain statistical manipulations of the data. Faceting can be used to replicate the plot using subsets of your data. Here we give a few examples to help you get started.

Returning to your October SOI values. To create a histogram with a bin width of one standard deviation (units of SOI), type

```
> require(ggplot2)
> qplot(Oct, data=SOI, geom="histogram", binwidth=1)
```

The `geom` argument (short for geometric object) represents what you see on the plot, here a histogram. The default geometric objects are points and lines.



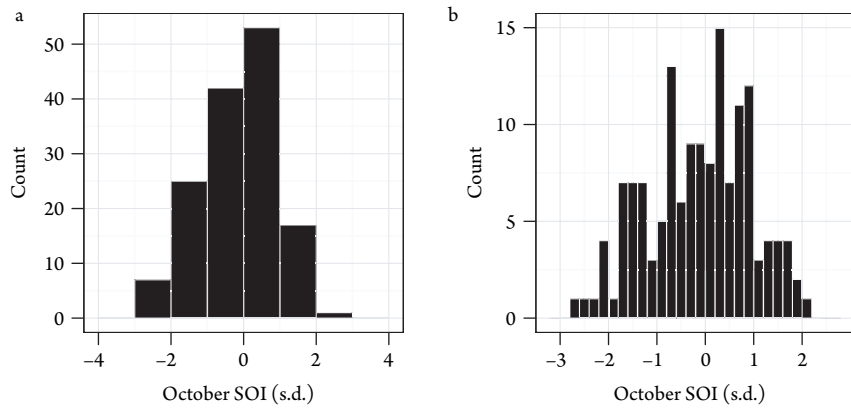


Figure 5.19 Histograms of October SOI.

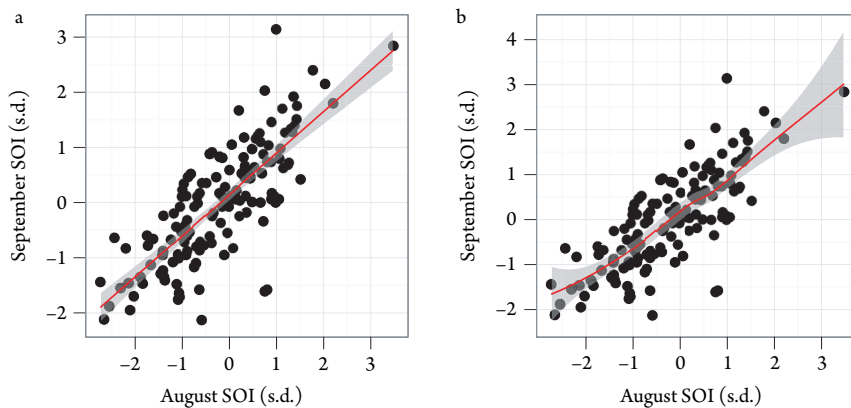


Figure 5.20 Scatter plots of August and September SOI.

Figure 5.19 shows histograms of the October SOI for two different bin widths. Note the use of grids and a background gray shade. This can be changed with the `theme_set` function.

You create a scatter plot using the same `qplot` function and in the same way as `plot`. Here you specify the data with an argument. The default geometric object in this case is the point.

```
> qplot(Aug, Sep, data=SOI)
```

You add a smoothing function (an example of a statistical manipulation of your data) with `smooth` as a character string in the `geom` argument.

```
> qplot(Aug, Sep, data=SOI, geom=c("point", "smooth"))
```

The default method for smoothing is local regression. You can change this to a linear regression by specifying `method="lm"`. Scatter plots with both types of smoothers are shown in Figure 5.20. The graph on the left uses the default local smoothing and

the graph on the right uses a linear regression. The `geom` plots the points and adds a best-fit line through them. The line is drawn by connecting predictions of the best-fit model at a set of equally spaced values of the explanatory variable (here August SOI) over the range of data values. A 95 percent confidence band about the prediction line is included.

Plots are built layer by layer. Layers are regular R objects and so can be stored as variables. This makes it easy for you to write clean code with a minimal amount of duplication. For instance, a set of plots can be enhanced by adding new data as a separate layer. As an example, here is code to produce the left plot in Figure 5.20.

```
> bestfit = geom_smooth(method="lm", color='red')
> pts = qplot(Aug, Sep, data=SOI)
> pts + bestfit
```

The `bestfit` layer is created and saved as a **geom** object and the `pts` layer is created from the `qplot` function. The two layers are added and then rendered to your graphics device in the third line of code.

Finally, consider again the NAO time series object you created in §5.2. You create a vector of times at which the series was sampled using the `times` function. Here you use the line **geom** instead of the default point.

```
> tm = time(nao.ts)
> qplot(tm, nao.ts, geom="line")
```

Results are shown in Figure 5.21. The values fluctuate widely from one month to the next, but there is no long-term trend. A local regression smoother (`geom_smooth`) using a span of 10 percent of the data indicates a tendency for a greater number of negative NAO values since the start of the twenty-first century.

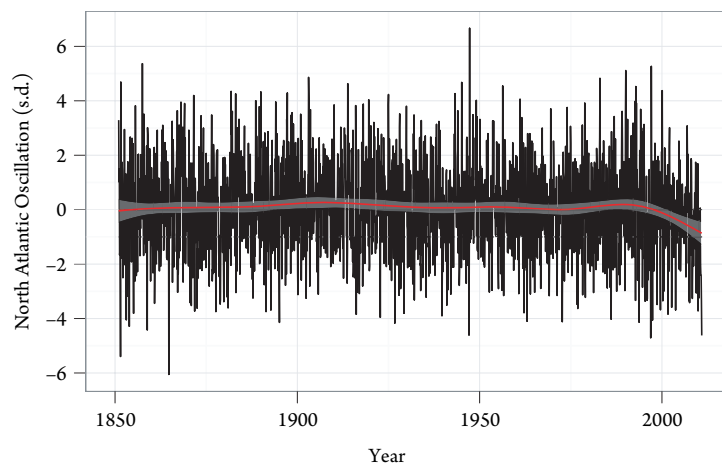


Figure 5.21 Time series of the monthly NAO. The red line is a local smoother.

As with the `plot` function, the first two arguments to `qplot` are the ordinate and abscissa data vectors, but you can use the optional argument `data` to specify column names in a data frame. The `ggplot` function, which allows even greater flexibility, accepts only data frames. Functions in **plyr** and **reshape** packages help you create data frames from other data objects (Teetor, 2011).

### 5.6.3 ggmap

The **ggmap** package (Kellie and Wickham, 2012) extends the grammar of graphics to maps. The function `ggmap` queries the Google Maps server or OpenStreetMap server for a map at a specified location and zoom. For example, to grab a map of Tallahassee, Florida, type

```
> require(ggmap)
> Tally = ggmap(location = "Tallahassee", zoom=13)
> str(Tally)
```

The result is an object of class `ggmap` with a matrix (640 × 640) of character strings specifying the fill color for each raster.

The level of zoom ranges from 0 for the entire world to 19 for the individual city blocks highest. The default zoom is 10. The default map type (`maptype`) is terrain with options for “roadmap”, “mobile”, “hybrid”, among others. To plot the map on your graphics device, type

```
> ggmapview(Tally)
```

To determine a center for your map, you use the `geocode` function to get a location. For example, to determine the location of Florida State University, type.

```
> geocode("Florida State University")
```

This chapter showed you how to produce graphs and maps with R. A good graph helps you understand your data and communicate your results. We began by looking at how to make bar charts, histograms, density plots, scatter plots, and graphs involving time. We then looked at utilities for drawing maps and described the various types of spatial data. We showed you how to create coordinate reference systems and transform between them. We also showed you how to export your graphs and maps. We ended by taking a look at two additional graphics systems within R. You will get more practice with these tools as you work through the book.