CHAPTER 3

Exploratory Data Analysis for Time Series

We will break down our discussion of exploratory data analysis for time series into two distinct sections. First, we discuss the application of commonly used data methods on time series. In particular, we discuss how histograms, plotting, and group-by operations can be applied to time series data.

Second, we highlight fundamentally temporal methods for time series analysis—that is, methods developed specifically for time series data and that make sense only in the context of data points that have a temporal relationship with one another (rather than a cross-sectional one).

Familiar Methods

We begin by thinking about how to apply commonly used data exploration techniques to time series data sets. The process is the same as what you have performed on non-time-series data to start with.

You will want to know the columns that are available, their value ranges, and what logical units of measurement work best. You will want to address the same exploratory questions you would ask about any new data set, such as:

- Are any of the columns strongly correlated with one another?
- What is the overall mean of an interesting variable? What is its variance?

To answer these, you can use familiar techniques such as plotting, taking summary statistics, applying histograms, and using targeted scatter plots. You'll also want to answer explicitly time-oriented questions such as:

- What is the range of values you see, and do they vary by time period or some other logical unit of analysis?

- Does the data look consistent and uniformly measured, or does it suggest changes in either measurement or behavior over time?

To address these, you will want the previously mentioned methods—histograms, scatter plots, and summary statistics—to take into account the temporal axis. To implement this behavior, we will incorporate time into our statistics, as an axis in our graphs or as a group in group-by operations, such as those applied to histograms or scatter plots.

In the rest of this section, we will look at examples of exploratory data analysis with a few different kinds of time series data to explore how we can use traditional nontemporal methods directly or with time-specific modifications. To demonstrate these exploratory methods, we will use European stock markets data, a time series data set available in base R.

Group-by Operations

With time series data, there are groups just as there are with nontemporal data, so group-by-style thinking is still applicable. For example, in cross-sectional data you might take group means based on age, gender, or neighborhood. In time series analysis, you might find exploratory group-by operations useful, such as computing a monthly average or weekly medians. You can also combine temporal and nontemporal groups, such as the monthly average calorie intake per gender in a population, or the weekly median amount of sleep per age group in a set of hospital patients. There are many ways that grouping data can allow the temporal axis to interact with other relationships within the data. For example:

- You may want to analyze different time periods separately, if you have exogenous reasons to expect a regime change. Use your exploratory analysis to find sensible demarcations for different meaningful time periods.
- You may discover that data presented as a single series (such as when donations come in) is clearer when broken down into many parallel processes (such as individual donation timelines).

Plotting

Let's explore a typical time series provided by R, the `EuStockMarkets` data set. To get a sense of it, we can look at the head of the data set:

```
## R
> head(EuStockMarkets)
        DAX   SMI    CAC    FTSE
[1,] 1628.75 1678.1 1772.8 2443.6
[2,] 1613.63 1688.5 1750.5 2460.2
```

```
[3,] 1606.51 1678.6 1718.0 2448.2
[4,] 1621.04 1684.1 1708.1 2470.4
[5,] 1618.16 1686.6 1723.1 2484.7
[6,] 1610.61 1671.6 1714.3 2466.8
```

This is a built-in data set in base R, and it contains the daily closing prices of four major European stock indexes from 1991 to 1998. The data set only includes business days.

This data set is better prepared than the ones we have looked at so far because it's already nicely formatted and sampled for us. We don't need to worry about missing values, time zones, or incorrect measurements, so we can jump right into exploratory data analysis.

Our first steps for looking at this data will be very similar to a non-time-based time series, although we have a simpler option with temporal data than we would with cross-sectional data. We can plot each value individually, and such a plot makes sense. Contrast this with cross-sectional data, where a plot of any individual feature against its index in the data set will only reveal, if anything, the order in which data was entered in a data table or returned from a SQL server (in other words, an arbitrary order), but nothing about the underlying situation. However, in the case of time series, plotting is quite informative, as you can see in Figure 3-1.

```
## R
> plot(EuStockMarkets)
```

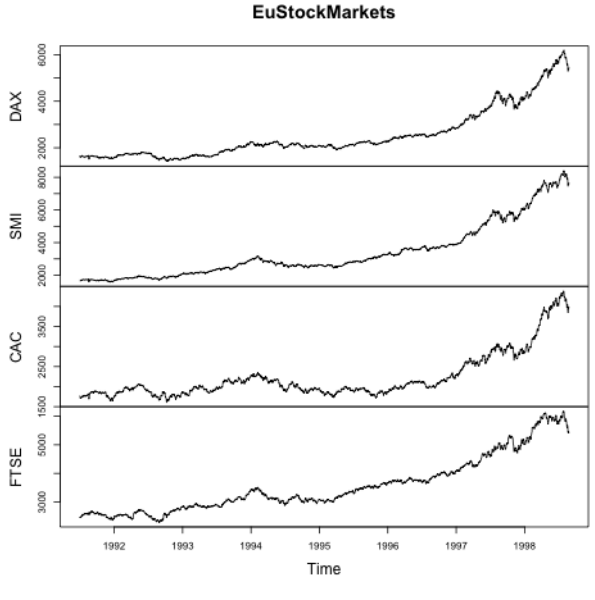

Figure 3-1. A simple plot of the time series data.

Notice that the image is automatically segmented into different time series with a simple `plot()` command. In fact, as shown here, we are using an R `mts` object (we would be using a `ts` object if there were only one time series, but we have several in parallel):

```
## R
> class(EuStockMarkets)
[1] "mts"    "ts"     "matrix"
```

A number of popular packages make heavy use of `ts` objects and derivative classes. These objects come with nice automatic calls to appropriate plotting functions, as we see in the previous example where a simple call to `plot()` created a nicely labeled, multipanel plot. `ts` objects also have a few convenience functions:

- `frequency` for finding out the yearly frequency of the data:

```
## R
> frequency(EuStockMarkets)
[1] 260
```

- `start` and `end` to find the first and last time represented in the series:

```
## R
> start(EuStockMarkets)
[1] 1991  130
> end(EuStockMarkets)
1] 1998  169
```

- `window` to take a temporal section of the data:

```
## R
> window(EuStockMarkets, start = 1997, end = 1998)
Time Series:
Start = c(1997, 1)
End = c(1998, 1)
Frequency = 260         DAX SMI CAC   FTSE
1997.000 2844.09 3869.8 2289.6 4092.5
1997.004 2844.09 3869.8 2289.6 4092.5
1997.008 2844.09 3869.8 2303.8 4092.5
...
1997.988 4162.92 6115.1 2894.5 5168.3
1997.992 4055.35 5989.9 2822.9 5020.2
1997.996 4125.54 6049.3 2869.7 5018.2
1998.000 4132.79 6044.7 2858.1 5049.8
```

There are pluses and minuses to the `ts` class. As mentioned earlier, `ts` and its derivative classes are used in many time series packages. Also, automatic setup of plotting parameters can be helpful. However, indexing can sometimes be tricky, and the

process of accessing subsections of data with `window` can feel cumbersome over time. In this book, you will see several ways of containing and accessing time series data, and it will be up to you to choose the most convenient for your use cases.

Histograms

Let's continue our comparison of exploratory data analysis in time series and non-time-series data. We'd like to, for example, take a histogram of the data, just as with most exploratory data analysis. We add a wrinkle by also taking a histogram of the differenced data because we want to use our time axis (see Figure 3-2):

```
## R
> hist(     EuStockMarkets[, "SMI"], 30)
> hist(diff(EuStockMarkets[, "SMI"], 30))
```

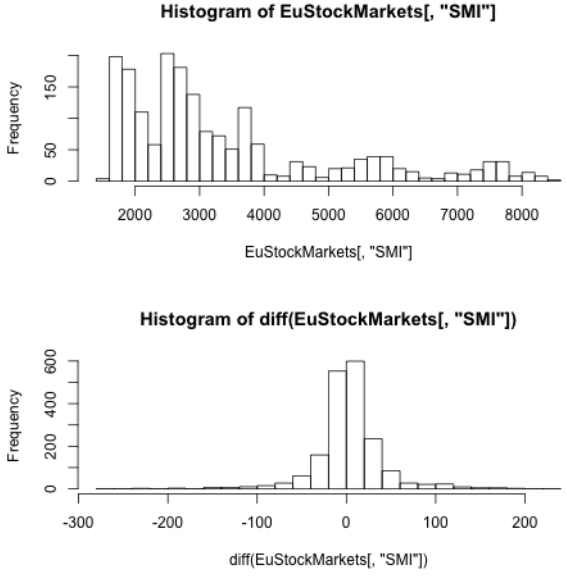

Figure 3-2. The histogram of the untransformed data (top) is extremely wide and does not show a normal distribution. This is to be expected given that the underlying data has a trend. We difference the data to remove the trend, and this transforms the data to a more normally shaped distribution (bottom).

In a time series context, a `hist()` of the difference of the data is often more interesting than a `hist()` of the untransformed data. After all, in a time series context, often (and particularly in finance) what is most interesting is how a value changes from one measurement to the next rather than the value's actual measurement. This is particularly true for plotting, because taking the histogram of data with a trend in it does not produce a very informative visualization.

Notice the new information we get from the histogram of the differenced series. While the original plots of the stocks in the previous section painted a very rosy economic picture with stocks inexorably rising, this histogram shows us the day-to-day experience of someone following the stocks. The histogram of the difference tells us that the value of the time series has gone both up (positive difference values) and down (negative difference values) about the same amount over time. The stock indexes did not go up and down exactly the same amount over time, because we know the stock indexes do have an increasing trend. However, we can see from this histogram that just a slight bias in favor of positive over negative differences is what accounts for that trend.

This is a good first example of why we need to pay attention to temporal scale when sampling, summarizing, and asking questions of data. Whether performance looks great (long-term plot) or just so-so (histogram of difference plots) will depend on what our temporal scale is: do we care about the day-to-day, or do we have a longer-term horizon? If we work in an organization that trades stock to make an annual profit, we need to think about the short-term experience we will have every time we report to our bosses a "diff" that leans toward negative values. However, if we are a large institutional investor—perhaps a university or a hospital—we may be able to afford to take the long view, which expects an upward climb. This latter scenario has its own challenges: how do we take long enough to maximize our profit but not so long that the party is over? These are the questions that keep the financial industry humming along with time series research and predictions.

Scatter Plots

The traditional method of using scatter plots is just as useful for time series data as it is for other kinds of data. We can use scatter plots to determine both how two stocks are linked at a specific time and how their price shifts are related over time.

In this example, we plot both cases (see Figure 3-3):

- The values of two different stocks over time
- The values of the daily changes in these two stocks over time (via differencing) with R's `diff()` function

```
## R
> plot(     EuStockMarkets[, "SMI"],       EuStockMarkets[, "DAX"])
> plot(diff(EuStockMarkets[, "SMI"]), diff(EuStockMarkets[, "DAX"]))
```

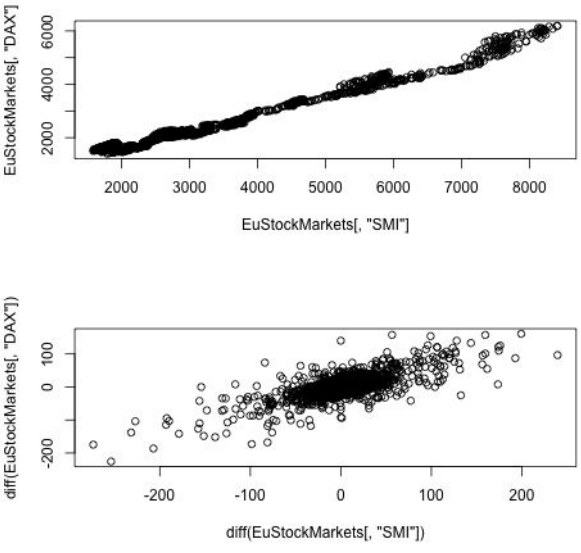

Figure 3-3. Simple scatter plots of two stock indexes suggest strong correlations. However, there are reasons to view these plots with suspicion.

As we've already seen, the actual values are less informative than the differences between adjacent time points, so we have plotted the diffs in a second scatter plot. These look like very strong correlations, but the relationships are not as strong as they appear. (To delve deeper, skip ahead to "Spurious Correlations" on page 102.)

The apparent correlations in Figure 3-3 are interesting, but even if they are true correlations (and there is reason to doubt they are), these are not correlations we can monetize as stock traders. By the time we know whether a stock is going up or down, the stocks it is correlated with will have also gone up or down, since we are taking correlations of values at identical time points. What we need to do is find out whether the change in one stock earlier in time can predict the change in another stock later in time. To do this, we shift one of the differences of the stocks back by 1 before looking at the scatter plot. Read the following code carefully; notice that we are still differencing, but now we are also applying a lag to one of the differenced time series (see Figure 3-4):

```
## R
> plot(lag(diff(EuStockMarkets[, "SMI"]), 1),
           diff(EuStockMarkets[, "DAX"]))
```

The lines in these code examples are easy to read because each section is aligned. It can be tempting to write long lines of unreadable code, particularly in a functional programming language such as R or Python, but you should avoid this whenever possible!

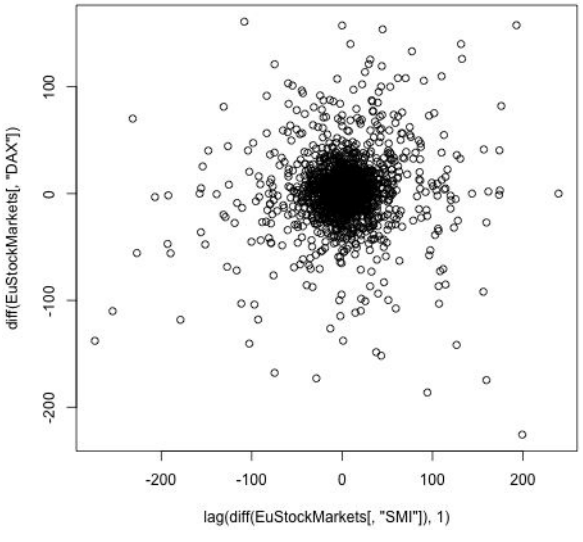

Figure 3-4. The correlation between the stocks disappears as soon as we put in a time lag, indicating that SMI does not appear to predict DAX.

This result tells us a number of important things:

- With time series data, while we may use the same exploratory techniques as with non-time-series data, mindless application won't work. We need to think about how to exploit the same techniques but with reshaped data.
- Oftentimes it's the relationship between data at different points or the change over time that is most informative about how your data behaves.
- The plot in Figure 3-4 shows why it can be difficult to be a stock trader. If you are a passive investor and wait it out, you might benefit from the long-term rising trend. However, if you are trying to make predictions about the future, you can see that it's not easy!

R's `lag()` function may not move data in the temporal direction you expect. The `lag()` function is a forward time shift. This is important to remember, as you wouldn't want to move data in the wrong temporal direction given a specific use case, and both forward and backward shifts in time are reasonable for different use cases.

Time Series–Specific Exploratory Methods

Several methods of analyzing time series data focus on relations of values at different times in the same series, and you likely won't have seen these before if you haven't worked with time series data. In the rest of this chapter, we walk through a few concepts and related techniques used to classify time series.

The concepts we will explore are:

Stationarity
: What it means for a time series to be stationary and a statistical test for stationarity

Self-correlation
: What it means to say that a time series correlates with itself and what such a correlation indicates about the underlying dynamics of the time series

Spurious correlations
: What it means for a correlation to be spurious and when you should expect to run into spurious correlations

The methods we will learn to apply are:

- Rolling and expanding window functions
- Self-correlation functions
  - The autocorrelation function
  - The partial autocorrelation function

We will cover the concepts and their resulting methods in order from stationarity to self-correlations to spurious correlations. Before we dive into the specifics, let's discuss the logic behind this particular ordering.

The first question you will likely ask about a time series is whether it appears to reflect a system that is "stable" or one that is constantly changing. The level of stability, or *stationarity*, is important to assess because we need to know how much we should expect the system's long-term past behavior to reflect its long-term future behavior. Once we have assessed the "stability" (this word is not used in a technical sense here) of a time series, we try to determine whether there are internal dynamics

in that series (seasonal changes, for example). That is, we are looking for *self-correlations*, answering the fundamental question of how tightly past data, distant or recent, predicts future data. Finally, when we think we have found certain behavioral dynamics within the system, we need to make sure we are not identifying relationships based on dynamics that do not in any way imply the causal relationships we wish to discover; hence, we must look for *spurious correlations*.

Understanding Stationarity

Many traditional statistical time series models rely on a time series being stationary. Generally speaking, a stationary time series is one that has fairly stable statistical properties over time, particularly with respect to mean and variance. This seems relatively straightforward.

Nonetheless, stationarity can be a slippery concept, particularly when applied to real time series data. It is both too intuitive and too easy to fool yourself into relying on your natural intuition. We will walk through the concept both intuitively and with a somewhat formal definition before discussing a common test for stationarity and practical details for how to apply the concept.

Intuition

A stationary time series is one in which a time series measurement reflects a system in a steady state. Sometimes it is difficult to assert what exactly this means, and it can be easier to rule things out as *not* being stationary rather than saying something *is* stationary. An easy example of data that is not stationary is the airline passengers data set we examined in Chapter 2, which is plotted in Figure 3-5 (as a reminder, it is available in R as `AirPassengers` and also widely available for download on the internet).

There are several traits that show this process is not stationary. First, the mean value is increasing over time, rather than remaining steady. Second, the distance between peak and trough on a yearly basis is growing, so the variance of the process is increasing over time. Third, the process displays strong seasonal behavior, the antithesis of stationarity.

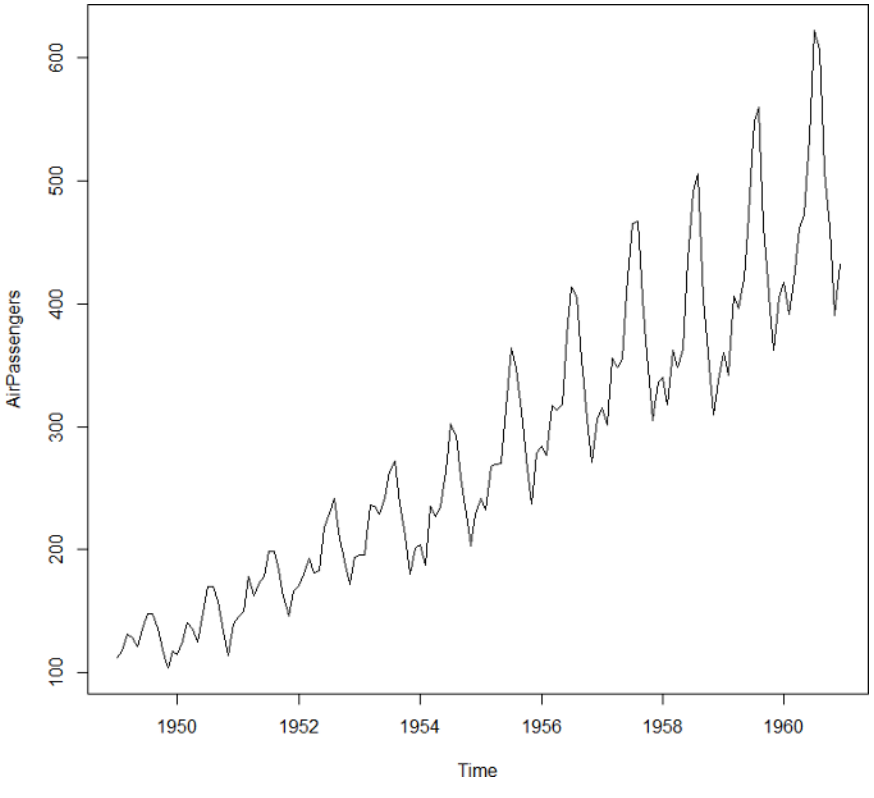

Figure 3-5. The airline passengers data set offers a clear example of a time series that is not stationary. As time passes, both the mean and the variance of the data are changing. We also see evidence of seasonality, which intrinsically reflects a nonstationary process.

Stationary definition and the Augmented Dickey–Fuller test

A simple definition of a stationary process is the following: a process is stationary if for all possible lags, k, the distribution of y_t, y_{t+1},..., y_{t+k}, does not depend on t.

Statistical tests for stationarity often come down to the question of whether there is a unit root—that is, whether 1 is a solution of the process's characteristic equation.[1] A linear time series is nonstationary if there is a unit root, although lack of a unit root does not prove stationarity. Addressing stationarity as a general question remains tricky, and determining whether a process has a unit root remains a current area of research.

1 If this doesn't ring a bell, don't worry about it. You can read more in "More Resources" on page 117, should you wish to pursue the topic.

Nonetheless, a simple intuition for what a unit root is can be gleaned from the example of a random walk:

$$y_t = \phi \times y_{t-1} + e_t$$

In this process, the value of a time series at a given time is a function of its value at the immediately preceding time and some random error. If ϕ is equal to 1, the series has a unit root, will "run away," and will not be stationary. Interesting to note is that the series not being stationary does not mean it has to have a trend. A random walk is a good example of a nonstationary time series that does not have an underlying trend.[2]

Tests for determining whether a process is stationary are called *hypothesis tests*. The Augmented Dickey–Fuller (ADF) test is the most commonly used metric to assess a time series for stationarity problems. This test posits a null hypothesis that a unit root is present in a time series. Depending on the results of the test, this null hypothesis can be rejected for a specified significance level, meaning the presence of a unit root test can be rejected at a given significance level.

Note that tests for stationarity focus on whether the mean of a series is changing. The variance is handled by transformations rather than formally tested. The test of whether a series is stationary is thus a test of whether a series is integrated. An integrated series of order d is a series that must be differenced d times to become stationary.

The framing of the Dickey–Fuller test is as follows:

$$\Delta y_t = y_t - y_{t-1} = (\phi - 1) \times y_{t-1} + \epsilon_t$$

Then the test of whether $\phi = 1$ is a simple t-test of whether the parameter on the lagged y_{t-1} is equal to 0. The difference that the ADF test makes is to account for more lags, so that the underlying model takes higher-order dynamics into account, which can be written as a series of differenced lags:

$$Y_t - \phi_1 \times y_{t-1} - \phi_2 \times y_{t-2} \ldots = \epsilon_t$$

This requires somewhat more algebra to write as a series of differenced lags, and the expected distribution against which to test the null hypothesis is somewhat different

2 A given sample time series process produced by a random walk can appear to have a trend, and this inspires much debate, particularly in analyses of stock-price time series.

compared to the original Dickey–Fuller test. The ADF test is the most widely presented test for stationarity in the time series literature.

Unfortunately, these tests are far from a panacea to your stationarity problems for a number of reasons:

- These tests have low power in distinguishing *near* unit roots from unit roots.
- With low sample size, false positives for unit roots are fairly common.
- Most tests do not test for or against all kinds of problems that can lead to a nonstationary time series. For example, some times will look specifically to testing whether the mean or the variance (but not both) is stationary. Other tests will look more generally at the overall distribution. It is important to understand the limits of the test applied when using it and to ensure that the limits are consistent with your beliefs about your data.

Setting an Alternate Null Hypothesis: KPSS Test

While the ADF test posits a null hypothesis of a unit root, the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test posits a null hypothesis of a stationary process. Unlike the ADF, the KPSS is not available in base R, but is still widely implemented. There are some nuances as to what these tests are for and how to use them properly; these are beyond the scope of this text but widely discussed online.

In practice

Stationarity matters in practice for a number of reasons. First, a large number of models assume a stationary process, such as traditional models with known strengths and statistical models. We will cover these classes of models in Chapter 6.

A broader point is that a model of a time series that is not stationary will vary in its accuracy as the metrics of the time series vary. That is, if you are seeking a model to help you estimate the mean of a time series with a nonstationary mean and variance, the bias and error in your model will vary over time, at which point the value of your model becomes questionable.

It is often the case that a time series can be made stationary enough with a few simple transformations. A log transformation and a square root transformation are two popular options, particularly in the case of changing variance over time. Likewise, removing a trend is most commonly done by differencing. Sometimes a series must be differenced more than once. However, if you find yourself differencing too much (more than two or three times) it is unlikely that you can fix your stationarity problem with differencing.

Log or sqrt?

While a square root tends to be less computationally complex than a logarithm, you should explore both options. Think about the range of your data and how you want to compress large values rather than prematurely optimizing (that is, unduly pessimizing) your code and analysis.

Stationarity is not the only assumption forecasting models make. Another common but distinct assumption is the normality of the distribution of the input variables or predicted variable. In such cases, other transformations may be necessary. A common one is the Box Cox transformation, which is implemented in the R `forecast` package and in `scipy.stats` in Python. The transformation makes non-normally distributed data (skewed data) more normal. However, just because you can transform your data doesn't mean you should. Think carefully about the meaning of the distances between data points in your original data set before transforming them, and make sure, regardless of your task, that transformations preserve the most important information.

Transformations Have Assumptions

You might think you're choosing only transformations that don't make underlying assumptions (`log` and `sqrt` seem simple, right?), but think carefully about whether that's true.

As noted, log and square root transformations are commonly used to reduce the variance *of variance* over time. These make a number of assumptions. One is that your data will always be positive. Alternately, if you choose to shift your data before taking the square root or log, you are again either adding a bias or assuming it doesn't matter. Finally, when you take the square root or log, you make larger values less different from one another, effectively compressing the space between larger values but not between smaller values, de-emphasizing the differences between outliers. This may or may not be appropriate.

Applying Window Functions

Let's review the most important time series exploratory graphs, which you are likely to use for most initial time series analysis.

Rolling windows

A common function distinct to time series is a window function, which is any sort of function where you aggregate data either to compress it (as we saw in the case of downsampling in the previous chapter) or to smooth it (also discussed in Chapter 2).

In addition to the applications already discussed, smoothed data and window-aggregated data make for informative exploratory visualizations.

We can calculate a moving average and other calculations that involve some linear function of a series of points with base R's `filter()` function, as follows:

```
## R
> ## calculate a rolling average using the base R
> ## filter function
> x  <- rnorm(n = 100, mean = 0, sd = 10) + 1:100
> mn <- function(n) rep(1/n, n)

> plot(x, type = 'l',                lwd = 1)
> lines(filter(x, mn( 5)), col = 2, lwd = 3, lty = 2)
> lines(filter(x, mn(50)), col = 3, lwd = 3, lty = 3)
```

This code produces the graph in Figure 3-6.

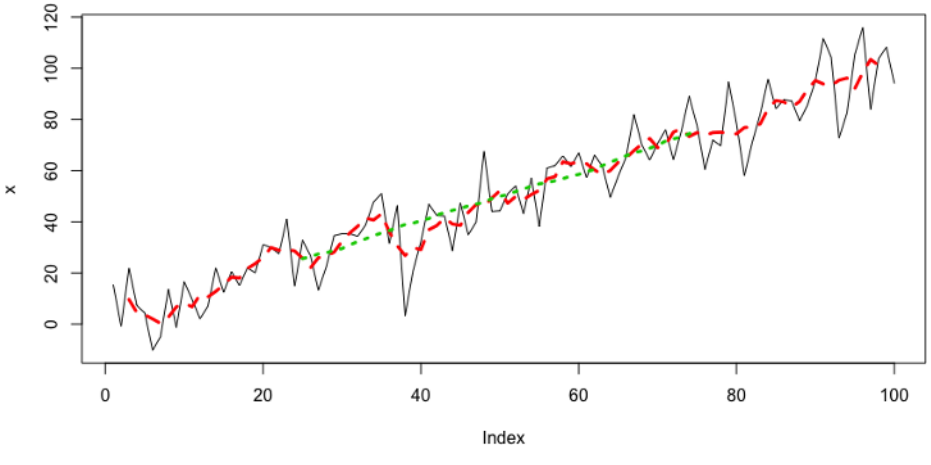

Figure 3-6. Two exploratory curves produced via a rolling mean smoothing. We might use these to look for a trend in particularly noisy data or to decide what sorts of deviations from linear behavior are interesting to investigate versus which are likely just noise.

If we are looking for functions that are not linear combinations of all points in the window, we can't use `filter()` because it relies on a linear transformation of the data. However, we can use `zoo`. The `rollapply()` function from the `zoo` package is quite handy (see Figure 3-7):

```
## R
> ## you can also do more 'custom' functionality
> require(zoo)

> f1 <- rollapply(zoo(x), 20, function(w) min(w),
```

```
>                 align = "left",  partial = TRUE)
> f2 <- rollapply(zoo(x), 20, function(w) min(w),
>                 align = "right", partial = TRUE)

> plot (x,           lwd = 1,         type = 'l')
> lines(f1, col = 2, lwd = 3, lty = 2)
> lines(f2, col = 3, lwd = 3, lty = 3)
```

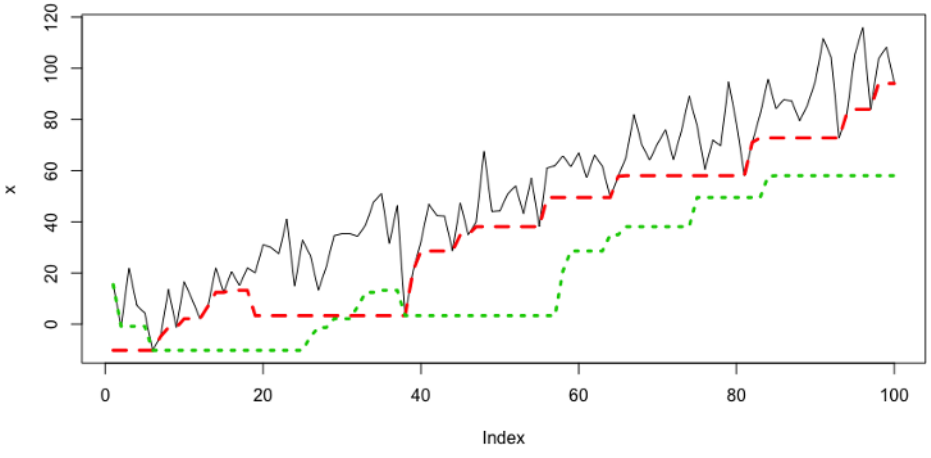

Figure 3-7. A rolling window minimum aligned either to the left (long dashes) or to the right (short dashes). The left alignment sees events in the future, whereas the right sees only events in the past. This is important to note for avoiding lookahead. However, sometimes a left alignment can be useful to ask exploratory questions such as "if I knew this in advance, would it even be useful?" Sometimes even a lookahead is not informative, which means that a particular variable is not informative. When knowing the future of a measure is not helpful, it's likely not a useful measure at any time in a time series context.

Use `zoo` objects in `zoo` functions. If you pass numeric vectors directly to `rollapply()`, the `align` argument will not take effect. You can confirm this by deleting the `zoo()` wrapper around `x` in the preceding code. You will see that the two curves are identical and, in fact, this is a silent failure, which is particularly dangerous in time series analysis because it can introduce an unintended lookahead. This is an example of why you need to sanity-check often, even with your exploratory data analysis. Unfortunately, silent failure is not unusual in many popular R packages and in other scripting languages, so stay alert!

It is also possible to "roll your own" for this kind of function, which can be a good idea to limit dependencies. In such cases, I'd strongly recommend starting with the source code from an existing widely used package, such as `zoo`. This is because, even for a univariate time series, there are a surprising number of corner cases to think about, such as how to treat `NA`s, and how to treat the beginning and ending of a series where you will have fewer points than the specified size of the window.

A Number of R Options for Time Series

Earlier in this chapter, we considered the `ts` class, which is available in base R to handle time series data. In this section we use `zoo` objects. You should also be aware of `xts` objects. Here is a brief summary of how `zoo` and `xts` objects improve on `ts` objects:

- A `ts` object assumes a regularly spaced complete time series and for this reason does not store individual time indices but instead only the start, end, and frequency of the time series.
- `ts` objects support a recurring cycle, such as years or months.
- `zoo` objects store timestamps as an index attribute, so that they do not require regularly spaced, periodic time series.
- `zoo` objects can be printed horizontally or vertically.
- The data part of a `zoo` object can be a vector or a matrix.
- `xts` objects are an extension of `zoo` objects with even more options.

Expanding windows

Expanding windows are less commonly used in time series analysis than rolling windows because their proper application is more limited. Expanding windows make sense only in cases where you are estimating a summary statistic that you believe is a stable process rather than evolving over time or oscillating significantly. An expanding window starts with a given minimum size, but as you progress into the time series, it grows to include every point up to a given time rather than only a finite and constant size.

An expanding window offers greater certainty in your estimate of test statistics over time, allowing you to benefit from being "deeper" into a particular time series. However, it works only if you hypothesize that your underlying system is stationary. It can help you maintain "online" summary statistics as you would have estimated them in real time as you gathered more information.

If you look into base R, you will realize that many functions already exist as implementations of an expanding window, such as `cummax` and `cummin`. You can also easily repurpose `cumsum` to create a cumulative mean. In the following plot, we show both an expanding window `max` and an expanding window `mean` (see Figure 3-8):

```
## R
> # expanding windows
> plot(x, type = 'l', lwd = 1)
> lines(cummax(x),              col = 2, lwd = 3, lty = 2) # max
> lines(cumsum(x)/1:length(x), col = 3, lwd = 3, lty = 3) # mean
```

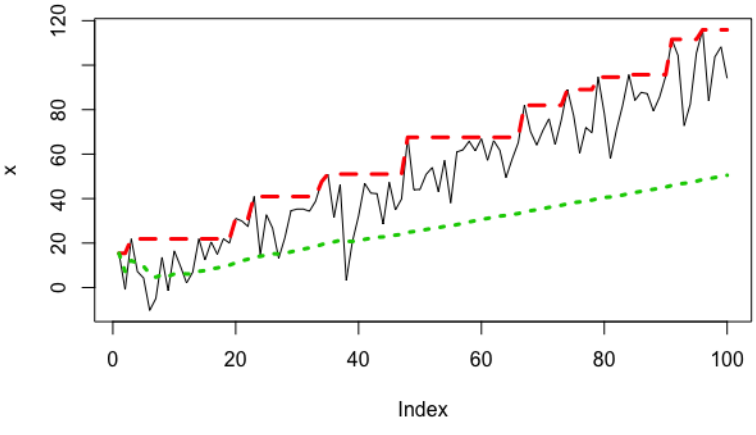

Figure 3-8. An expanding window "max" (long dashes) and an expanding window "mean" (short dashes). Use of an expanding window means that the max always reflects the global max up until that time, making it a monotonic function. Thanks to the long "memory" of the expanding window, the expanding window mean is lower than the rolling mean (Figure 3-7) because the underlying trend is less prominent in the expanding mean. Whether this is good, bad, or neutral depends on our assumptions and knowledge of the underlying system.

If you need a custom function with a rolling window, you can use `rollapply()` as we did for a rolling window. In this case, you need to specify a sequence of window sizes rather than a single scalar. Running the following code will produce an identical plot to the one in Figure 3-8, but this time it was created with `rollapply()` rather than built-in R functions:

```
## R
> plot(x, type = 'l', lwd = 1)
> lines(rollapply(zoo(x), seq_along(x), function(w) max(w),
>                         partial = TRUE, align = "right"),
>          col = 2, lwd = 3, lty = 2)
> lines(rollapply(zoo(x), seq_along(x), function(w) mean(w),
>                         partial = TRUE, align = "right"),
>          col = 3, lwd = 3, lty = 3)
```

Custom rolling functions

We only care about the option to apply a custom rolling function if we can actually imagine doing so. In practice, this is something you are likely to see when analyzing time series domains that have known underlying fundamental laws of behavior or useful heuristics that are necessary for proper analysis.

For example, we may be looking at windows that include a particular feature that is informative given domain knowledge. We might want to know when we are in a *monotonic* regime (say, blood sugar is increasing) versus an up-and-down regime that suggests instrumental noise rather than a trend. We could write a custom function for this scenario and apply it with either a moving or expanding window.

Understanding and Identifying Self-Correlation

At its most fundamental, self-correlation of a time series is the idea that a value in a time series at one given point in time may have a correlation to the value at another point in time. Note that "self-correlation" is being used here informally to describe a general idea rather than a technical one.

As an example of self correlation, if you take a yearly time series of daily temperature data, you may find that comparing May 15th of every year to August 15th of every year will give you some correlation, such that hotter May 15ths tend to correlate with hotter August 15ths (or tend to correlate with cooler August 15ths). You may feel you have learned a potentially interesting fact about the temperature system, indicating that there is a certain amount of long-term predictability. On the other hand, you may find the correlation closer to zero, in which case you will also have found something interesting, namely that knowing the temperature on May 15th does not alone give you any information about the likely range of temperatures on August 15th. That is self-correlation in an anecdotal nutshell.

From this simple example, we are going to expand into autocorrelation, which generalizes self-correlation by not anchoring to a specific point in time. In particular, autocorrelation asks the more general question of whether there is a correlation between any two points in a specific time series with a specific fixed distance between them. We'll look at this in more detail next, as well as final elaboration of partial autocorrelation.

The autocorrelation function

We begin with Wikipedia's excellent definition of autocorrelation:

> Autocorrelation, also known as serial correlation, is the correlation of a signal with a delayed copy of itself as a function of the delay. Informally, it is the similarity between observations as a function of the time lag between them.

Let's translate that into plainer English. Autocorrelation gives you an idea of how data points at different points in time are linearly related to one another as a function of their time difference.

The *autocorrelation function* (ACF) can be intuitively understood with plotting. We can plot it easily in R (see Figure 3-9):

```
## R
> x <- 1:100
> y <- sin(x * pi /3)
> plot(y, type = "b")
> acf(y)
```

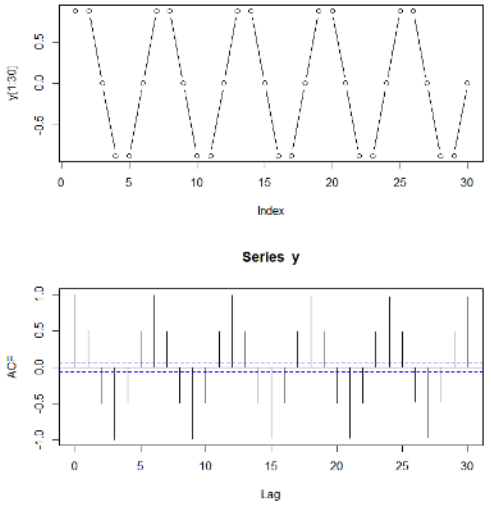

Figure 3-9. Plot of a sine function and its ACF.

Correlation in a Deterministic System

This sine series is a simple function and a fully determined system given a known input sequence. Nonetheless, we do not have a correlation of 1. Why is that? Inspect the series and think about what the ACF is calculating, and you will realize that for many values, the subsequent value can go either up or down depending on where you are in the cycle. If you know a few points in a row, you know which direction the process is going, but if you don't (as with the ACF, which is a 1:1 correlation measure), you will have a correlation of less than 1 because most values do not have a unique subsequent value but rather more than one. So, remember that a nonunitary correlation does not mean you necessarily have a probabilistic or noisy time series.

From the ACF, we see that points that have a lag between them of 0 have a correlation of 1 (this is true for every time series), whereas points separated by 1 lag have a correlation of 0.5. Points separated by 2 lags have a correlation of –0.5, and so on.

Calculating the ACF is straightforward. We can do it ourselves using `data.table`'s `shift()` function:

```
## R
> cor(y, shift(y, 1), use = "pairwise.complete.obs")
[1] 0.5000015
> cor(y, shift(y, 2), use = "pairwise.complete.obs")
[1] -0.5003747
```

Our calculations approximately match the graphical results in Figure 3-9.[3] While it is straightforward to calculate the ACF with custom code, it's usually better to use a prerolled version, such as R's `acf` function. There are several advantages to doing so:

- Automatic plotting with helpful labels
- A sensible (usually but not always) max number of lags for which to calculate ACF, as well as the option to override this max
- A graceful way to handle multivariate time series

There are a few important facts about the ACF, mathematically speaking:

- The ACF of a periodic function has the same periodicity as the original process. You can see this in the preceding sine example plots.
- The autocorrelation of the sum of periodic functions is the sum of the autocorrelations of each function separately. You can easily formulate an example of this with some simple code.
- All time series have an autocorrelation of 1 at lag 0.
- The autocorrelation of a sample of white noise will have a value of approximately 0 at all lags other than 0.
- The ACF is symmetric with respect to negative and positive lags, so only positive lags need to be considered explicitly. You can try plotting a manually calculated ACF to prove this.
- A statistical rule for determining a significant nonzero ACF estimate is given by a "critical region" with bounds at +/–1.96 × sqrt(n). This rule relies on a sufficiently large sample size and a finite variance for the process.

3 Sample correlations calculated with R's `cor()` function versus with R's `acf()` function will not match exactly because they use different divisors. For more information, see StackExchange.

The partial autocorrelation function

The partial autocorrelation function (PACF) can be trickier to understand than the ACF. The partial autocorrelation of a time series for a given lag is the partial correlation of the time series with itself at that lag given all the information between the two points in time.

It's easy to nod your head here and think that sounds reasonable. But what exactly does it mean to account for the information between two points in time? It means you need to compute a number of conditional correlations and subtract these out of the total correlation. Computing the PACF is not straightforward, and there are a variety of methods for estimating it. We will not discuss these here, but you can find discussions in the related R and Python documentation.

The PACF is easier to understand graphically than conceptually. Its utility is also much more obvious in a graph than in a discussion (see Figure 3-10):

```
## R
> y <- sin(x * pi /3)
> plot(y[1:30], type = "b")
> pacf(y)
```

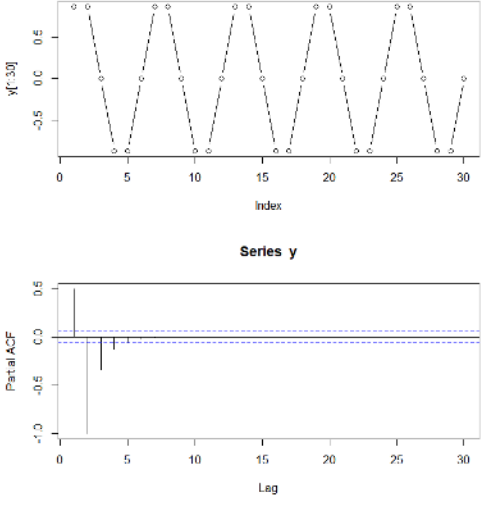

Figure 3-10. Plot and PACF of a seasonal, noiseless process.

For the case of a sine series, the PACF provides a striking contrast to the ACF. The PACF shows which data points are informative and which are harmonics of shorter time periods.

For a seasonal and noiseless process, such as the sine function, with period T, the same ACF value will be seen at T, 2T, 3T, and so on up to infinity. An ACF fails to weed out these redundant correlations. The PACF, on the other hand, reveals which correlations are "true" informative correlations for specific lags rather than redundancies. This is invaluable for knowing when we have collected enough information to get a sufficiently long window at a proper temporal scale for our data.

The critical region for the PACF is the same as for the ACF. The critical region has bounds at $\pm 1.96\sqrt{n}$. Any lags with calculated PACF values falling inside the critical region are effectively zero.

We have so far looked only at examples of perfectly noiseless single-frequency processes. Now we look at a slightly more complicated example. We'll consider the sum of two sine curves under no noise, low noise, and high noise conditions.

First, let's look at the plots with no noise, each individually (see Figure 3-11):

```
## R
> y1 <- sin(x * pi /3)
> plot(y1, type = "b")
> acf (y1)
> pacf(y1)

> y2 <- sin(x * pi /10)
> plot(y2, type = "b")
> acf (y2)
> pacf(y2)
```

The ACF of stationary data should drop to zero quickly. For nonstationary data the value at lag 1 is positive and large.

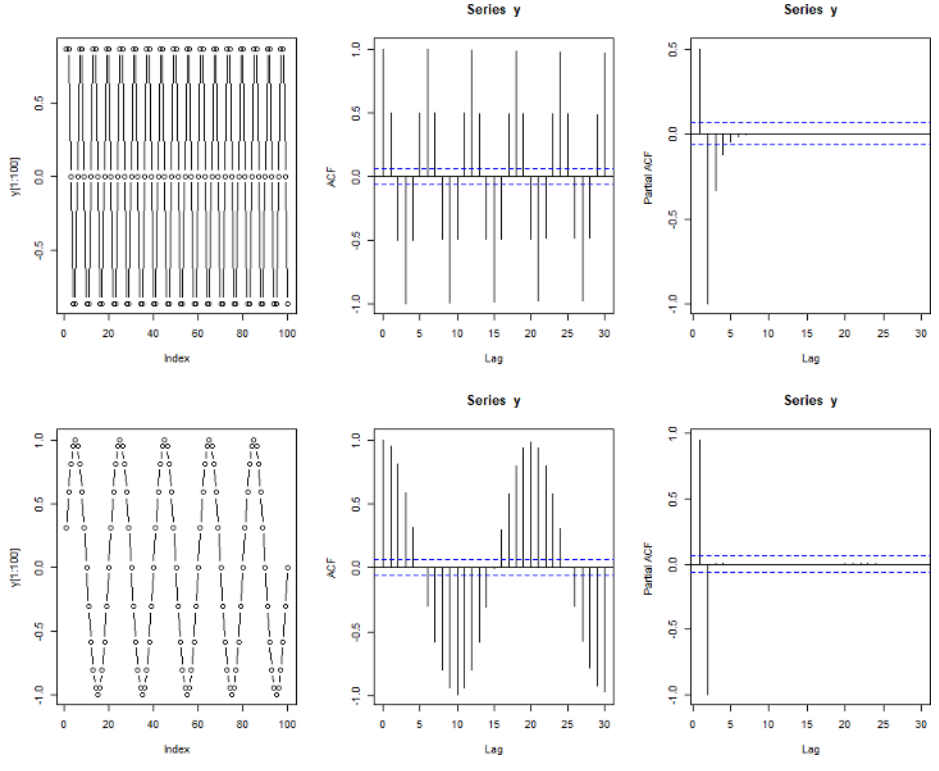

Figure 3-11. Plots of two sine functions, their ACF, and their PACF.

We combine the two series by summing and create the same plots for the summed series (see Figure 3-12):

```
## R
> y <- y1 + y2
> plot(y, type = "b")
> acf (y)
> pacf(y)
```

As we can see, our ACF plot is consistent with the aforementioned property; the ACF of the sum of two periodic series is the sum of the individual ACFs. You can see this most clearly by noticing the positive → negative → positive → negative sections of the ACF correlating to the more slowly oscillating ACF. Within these waves, you can see the faster fluctuation of the higher-frequency ACF.

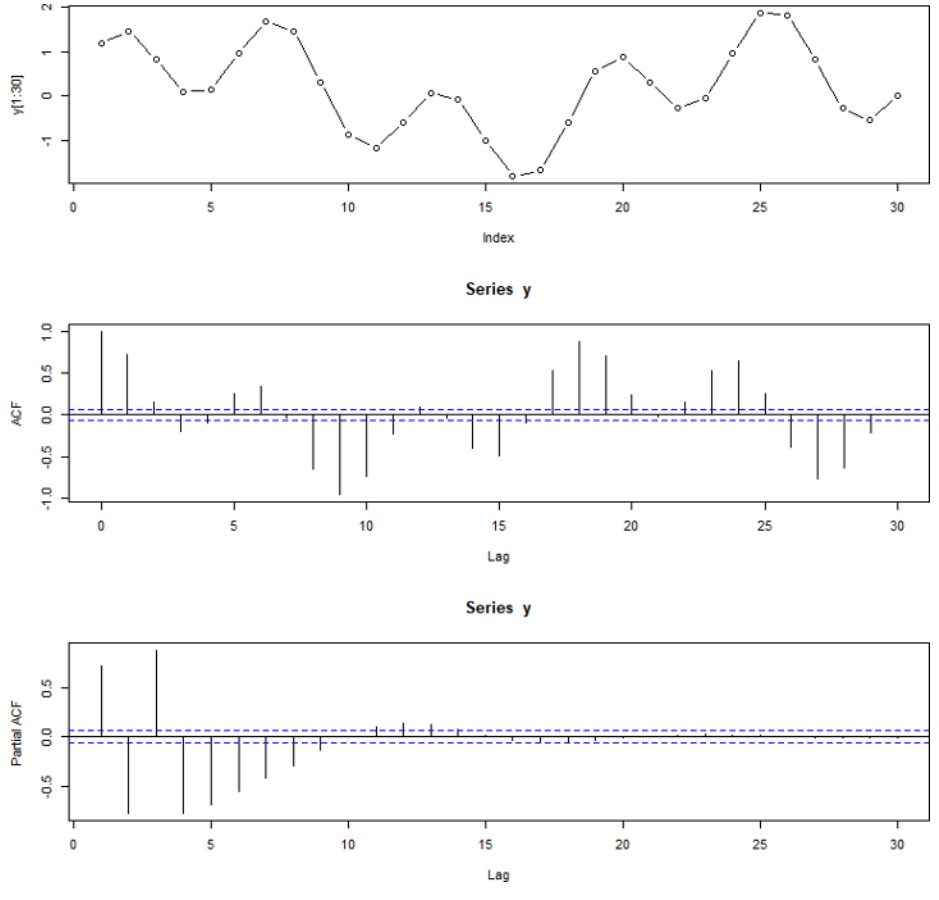

Figure 3-12. Plot, ACF, and PACF for the sum of the two sine series.

The PACF is not a straightforward sum of the PACF functions of the individual components. A PACF is simple enough to understand once it is calculated, but it's not so easy to generate or predict. This PACF indicates that partial autocorrelation is more substantial in the summed series than in either of the original series. That is, the correlation between points separated by a certain lag, when accounting for the values of the points between them, is more informative in the summed series than in the original series. This is related to the two different periods of the series, which result in any given point being less determined by the values of neighboring points since the location within the cycle of the two periods is less fixed now as the oscillations continue at different frequencies.

Let's look at the same situation, but with more noise (see Figure 3-13):

```
## R
> noise1 <- rnorm(100, sd = 0.05)
> noise2 <- rnorm(100, sd = 0.05)

> y1 <- y1 + noise1
> y2 <- y2 + noise2
> y  <- y1 + y2

> plot(y1, type = 'b')
> acf (y1)
> pacf(y1)

> plot(y2, type = 'b')
> acf (y2)
> pacf(y2)

> plot(y, type = 'b')
> acf (y)
> pacf(y)
```

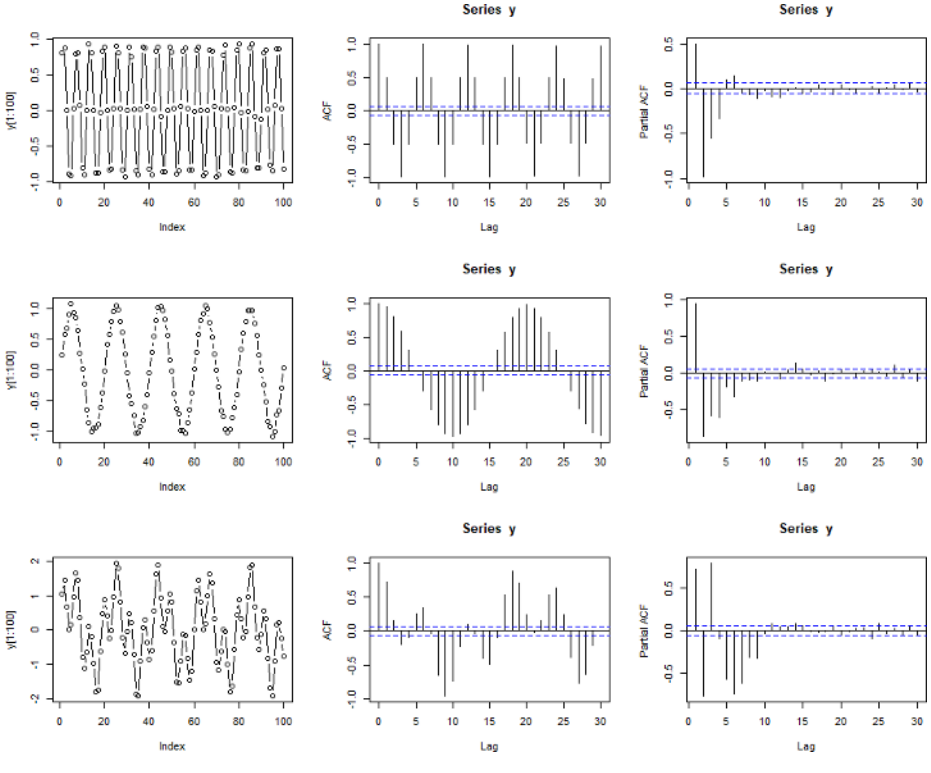

Figure 3-13. Plot, ACF, and PACF of two noisy sine processes and their sum.

Finally we add more noise to the time series so that the initial data itself does not even look particularly sine-like. (We omit the code example because it is the same as the previous example, simply with a larger `sd` parameter for `rnorm`.) We can see that this adds further interpretation difficulties, particularly to the PACF. The only difference between the plots in Figure 3-14 and the previous ones is a larger `sd` value for the `noise` variables.

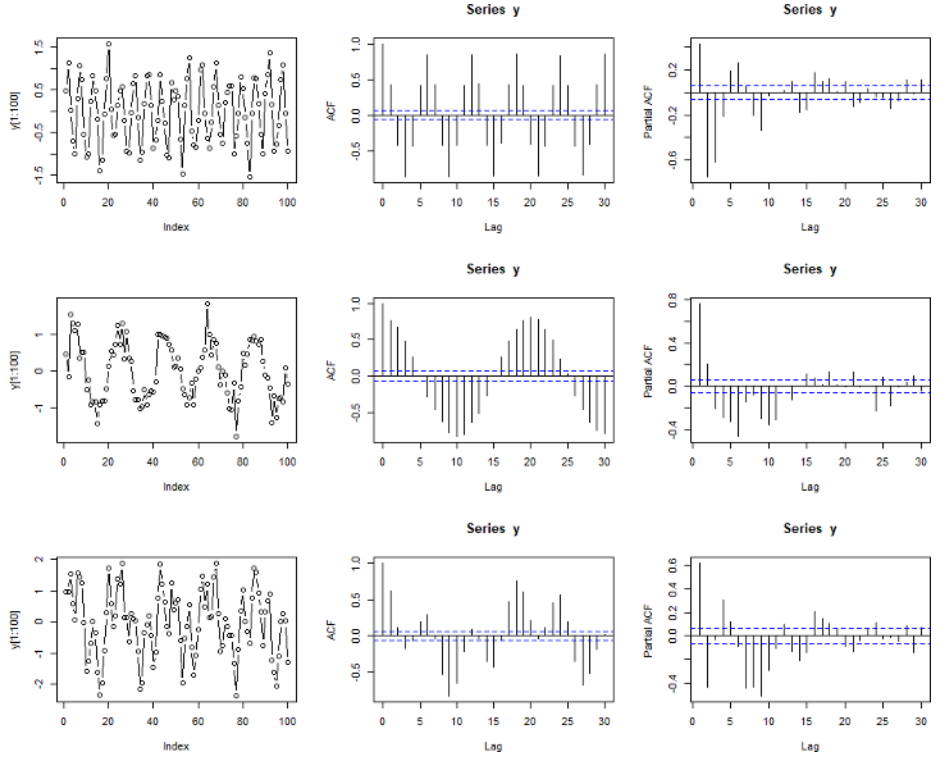

Figure 3-14. Plot, ACF, and PACF of very noisy sum of two sine processes.

Nonstationary Data

Let's consider how the ACF and PACF would look in the event of a series with a trend but no cycle (see Figure 3-15):

```
## R
> x <- 1:100
> plot(x)
> acf (x)
> pacf(x)
```

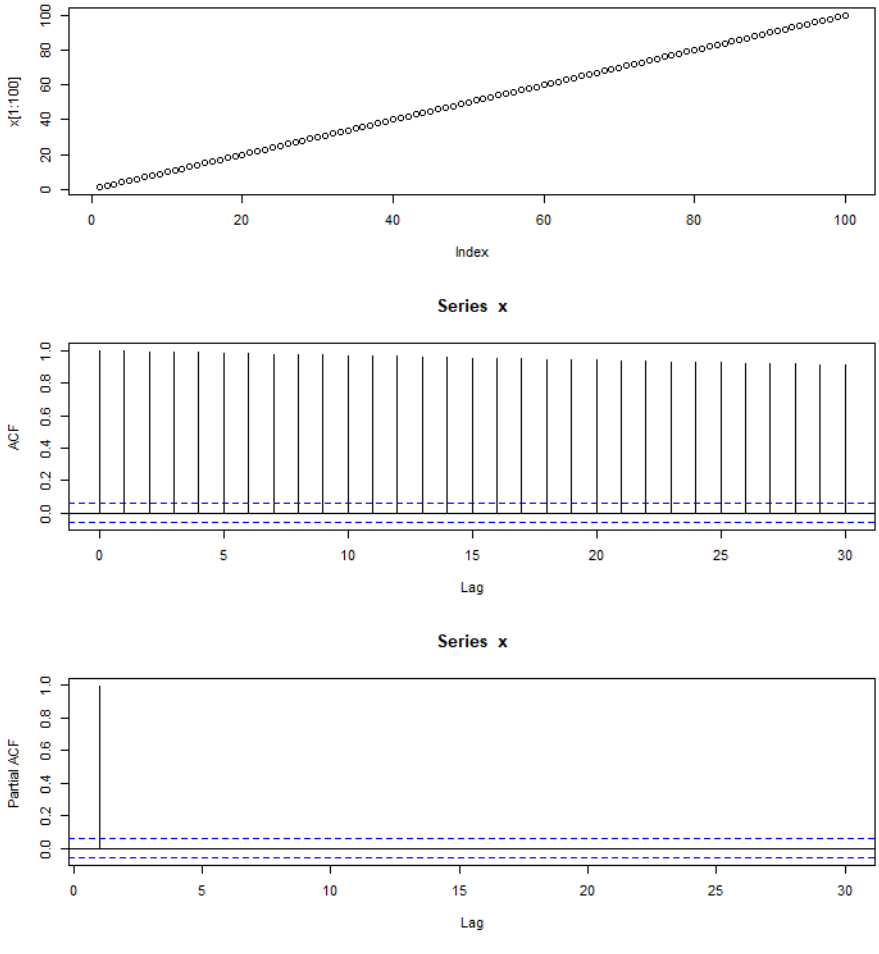

Figure 3-15. Plot, ACF, and PACF of a linearly trending process.

The ACF is not informative. It has a similar value for every lag, seemingly implying that all lags are equally correlated to the data. It's not clear what this means, and it's more a case of being able to compute *something*, not being able to make that computed quantity meaningful or sensible.

Luckily, the PACF is not as difficult to contemplate and gives us the information we need, which is that the only significant PACF correlation is at lag 1. Why is that? For a given point in time, once you know the point just before it, you know all the information the series can possibly give you about your point in time. This is due to the next point in time being 1 plus the old point.

Let's conclude with a look at the ACF and PACF of a real-world data set. Below, we examine the `AirPassengers` data. Given what we have seen so far, think about why the ACF has so many "critical" values (answer: it has a trend) and why the PACF has a critical value for a large lag (answer: the yearly seasonal cycle, which is identifiable even with a trend in the data).

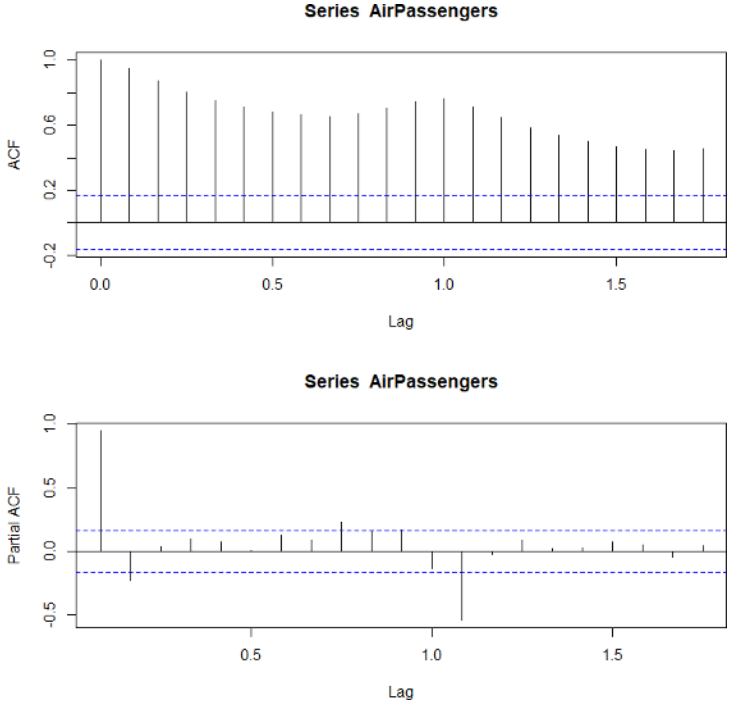

Figure 3-16. Plot of the ACF and PACF of the AirPassengers data. The lags here are not unit numbers because the lags are expressed as fractions of a year. This is because the AirPassengers data set comes in the form of a ts object, which has a built-in frequency that can be used for plotting (and other purposes).

Spurious Correlations

Those new to time series analysis will often begin with standard exploratory data practices, such as plotting two variables against each other and calculating their correlation. The new analyst will be very excited early on in the data exploration process when they notice what appears to be a very strong correlation and helpful relationship. They will continue looking and find other surprisingly high correlations; what an amazing system! They will wish they had started working with time series data sooner in their career. They will think to themselves, "All I do is win."

Then the analyst will do a little reading on time series analysis (best-case scenario), or present their findings to someone else and realize that it doesn't all make sense (not the best-case scenario). A skeptic will point out to the analyst that the correlations are too high. It can seem like any two values are related. More problems will emerge as the analyst reruns their analysis with other sets of variables and finds that they too have surprisingly high correlations. At some point it will become clear that there can't possibly be so many truly high correlations.

This trajectory is very much like the early history of econometrics. In the 19th century, when economists first began thinking of the idea of a business cycle, some of them went looking for external drivers of the cycle, such as sun spots (an 11-year cycle) or various meteorological cycles (such as a posited 4-year precipitation cycle). They invariably got very positive and strongly correlated results, even when they had no causal hypothesis to explain these results.

Many economists and statisticians remained skeptical, and rightly so. Udny Yule investigated the problem formally with a paper titled "Why Do We Sometimes Get Nonsense Correlations?", and an area of research was born and continues to give trouble and pleasure to academics. Spurious correlations remain an important problem to guard against, and they are hotly debated in litigation situations where one side asserts a relationship and the other side tries to discredit it. Similarly, one attempt to discredit climate change data relies on an argument that the correlation between increasing carbon emissions and warming global temperatures is a spurious correlation due to the trends in the two data sets (I do not find this argument convincing).

Economists have learned over time that data with an underlying trend is likely to produce spurious correlations. Here's a simple way to think about this: there is more information in a trending time series than in a stationary time series, so there are more opportunities for data points to move together.

In addition to trends, some other common features of time series can introduce spurious correlations:

- Seasonality—for example, think of a spurious correlation between hot dog consumption and death by drowning (summer).
- Level or slope shifts in data from regime changes over time (producing a dumbbell-like distribution with meaningless high correlation).
- Cumulatively summed quantities (this is a trick used in certain industries to make models or correlations look better than they are).

Cointegration

Cointegration refers to a real relationship between two time series. A commonly used example is a drunk pedestrian and their dog. Their individually measured walks might appear random taken alone, but they never stray too far from each other.

In the case of cointegration you will see high correlations. The difficulty will be in assessing whether two processes are cointegrated or whether you are seeing a spurious correlation because in both cases you will see surprisingly high correlations. The important difference is that there need not be any relationship in the case of a spurious correlation, whereas cointegrated time series are strongly related to one another.

There is a well-known blog (and now book) full of wonderful examples of spurious correlations, and I share one in Figure 3-17. Whenever you are tempted to think you have found a special and very strong relationship, make sure to check your data for obvious causes of trouble such as trends.

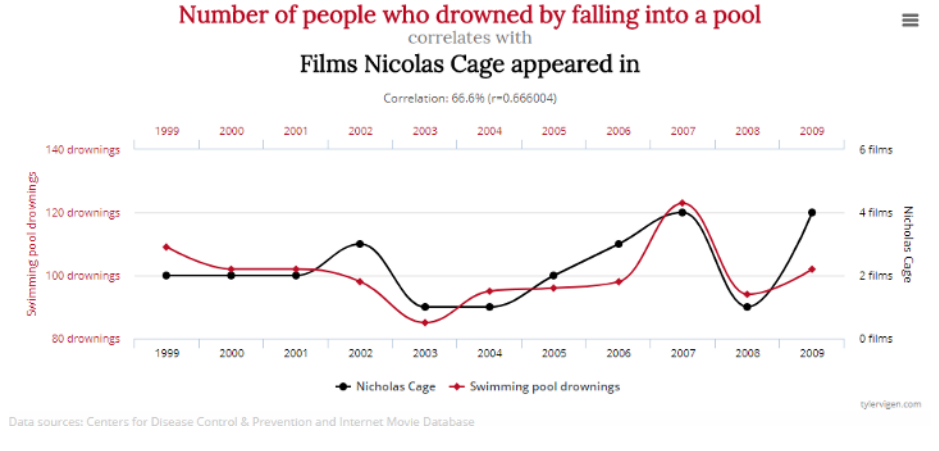

Figure 3-17. Some spurious correlations can look surprisingly convincing. This plot was taken from Tyler Vigen's website of spurious correlations.

Some Useful Visualizations

Graphs are fundamental to a thorough exploratory analysis of time series. You will certainly want to visualize data with respect to the temporal axis—preferably in a way that answers the general questions you have about the data set, such as the behavior of a particular variable or the overall temporal distribution of the data points.

Earlier in this chapter, we looked at some plotting techniques familiar to any data analyst, such as a plot of values against time or a scatter plot of different columns' values over time. In this final section on exploratory data analysis, we discuss several visualizations that are particularly helpful for offering novel insights about time series behavior.

We will look at visualizations of varying degrees of complexity:

- A one-dimensional visualization to understand the overall temporal distribution of individuals with a found time series we assembled in Chapter 2
- A two-dimensional histogram to understand the typical trajectory of a value over time in the case of many parallel measurements (say many years measured or many time series of the same phenomenon measured)
- A three-dimensional visualization where time can take up as many as two of the dimensions or as few as none of the dimensions, but still be implicitly present

1D Visualizations

In the cases of many units of measurement (many users, members, etc.) we consider multiple time series in parallel. It can be interesting to stack these visually, emphasizing individual units of analysis and their respective time frames. We ignore the values measured and rather take the existence of data over a given range as the information of interest. The time span itself becomes the unit of analysis. Here we use R's `timevis` package, but there are many other options available. We look at a small subset of the `donations` data we prepared in Chapter 2 (see Figure 3-18):

```
## R
> require(timevis)
> donations <- fread("donations.csv")
> d         <- donations[, .(min(timestamp), max(timestamp)), user]
> names(d)  <- c("content", "start", "end")
> d         <- d[start != end]
> timevis(d[sample(1:nrow(d), 20)])
```

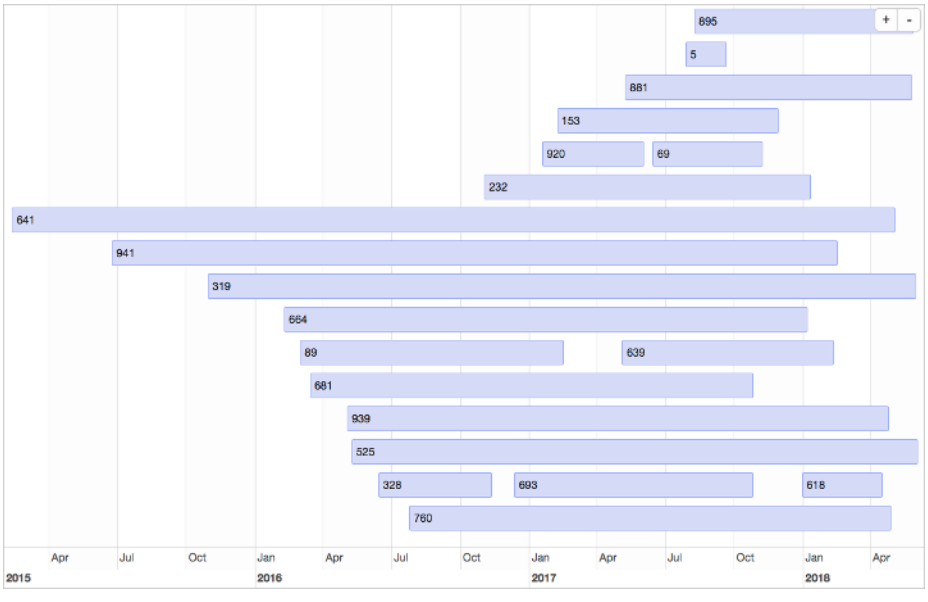

Figure 3-18. A Gantt chart of a random sample of data can offer some idea of the distribution of the range of "active" time periods for the users/donors.

The chart in Figure 3-18 helps us see that we probably have "busy" periods globally across the member population. We also glean some sense of the distribution of active donation spans in a member's "lifetime" in our organization.

Gantt charts have been used for over a century, most often for project management tasks. They came about independently in many different industries, and the idea is intuitive as soon as you see one. Despite the project management origins, Gantt charts can be useful in time series analysis where there are many independent actors, rather than a single process being measured. The plot in Figure 3-18 quickly answered questions I had about the relative overlap among the entire user base with respect to their donation history, a distribution I found difficult to understand when merely reading through the tabular data.

2D Visualizations

Now we'll use the `AirPassengers` data to see the seasonality and the trend, but we shouldn't think of time as linear. In particular, time happens on more than one axis. There is, of course, the axis of time going forward from day to day and year to year, but we can also consider laying time out along the axis of hour of the day or day of the week, and so on. In this way, we can more easily think about seasonality, such as certain behaviors happening at a certain time of the day or month of the year. We

think in particular about how to understand our data in a seasonal fashion rather than just according to linear, chronological time visualizations.

We extract the data from the `AirPassengers ts` object and put it into appropriate matrix form:

```
## R
> t(matrix(AirPassengers, nrow = 12, ncol = 12))
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]   112  118  132  129  121  135  148  148  136   119  104  118
[2,]   115  126  141  135  125  149  170  170  158   133  114  140
[3,]   145  150  178  163  172  178  199  199  184   162  146  166
[4,]   171  180  193  181  183  218  230  242  209   191  172  194
[5,]   196  196  236  235  229  243  264  272  237   211  180  201
[6,]   204  188  235  227  234  264  302  293  259   229  203  229
[7,]   242  233  267  269  270  315  364  347  312   274  237  278
[8,]   284  277  317  313  318  374  413  405  355   306  271  306
[9,]   315  301  356  348  355  422  465  467  404   347  305  336
[10,]  340  318  362  348  363  435  491  505  404   359  310  337
[11,]  360  342  406  396  420  472  548  559  463   407  362  405
[12,]  417  391  419  461  472  535  622  606  508   461  390  432
```

Notice we had to transpose the data so that it will line up as presented by the `ts` object.

Column Major Versus Row Major

R is column major by default, which is unusual and different from Python's `NumPy` (row major) and also different from most SQL databases. It's good to be aware of what behaviors are default and available in a given language, not just for display purposes but to think about how to manage and access memory effectively later on.

We plot each year on a set of axes that reflect the progression of the months across the year (see Figure 3-19):

```
## R
> colors <- c("green",  "red",         "pink",   "blue",
>             "yellow","lightsalmon", "black",  "gray",
>             "cyan",  "lightblue",   "maroon", "purple")
> matplot(matrix(AirPassengers, nrow = 12, ncol = 12),
>         type = 'l', col = colors,  lty = 1, lwd = 2.5,
>         xaxt = "n", ylab = "Passenger Count")
> legend("topleft", legend = 1949:1960, lty = 1, lwd = 2.5,
>        col = colors)
> axis(1, at = 1:12, labels = c("Jan", "Feb", "Mar", "Apr",
>                               "May", "Jun", "Jul", "Aug",
>                               "Sep", "Oct", "Nov", "Dec"))
```

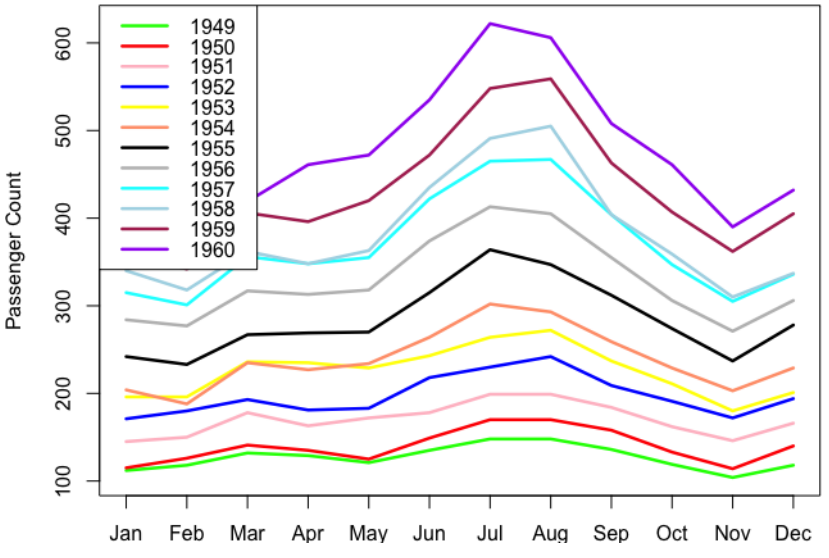

Figure 3-19. Per-year month-by-month counts.[4]

We can produce the same plot more easily with the `forecast` package (see Figure 3-20):

```
## R
> require(forecast)
> seasonplot(AirPassengers)
```

The x-axis is month of the year for all years. Every year, the number of airline passengers peaks in July or August (months 7 and 8). We also see a local peak in March (month 3) most years. This graph can thus show us more details regarding the seasonality behavior.

4 Visit the GitHub repository to view the original figure, or plot it yourself for a more detailed look.

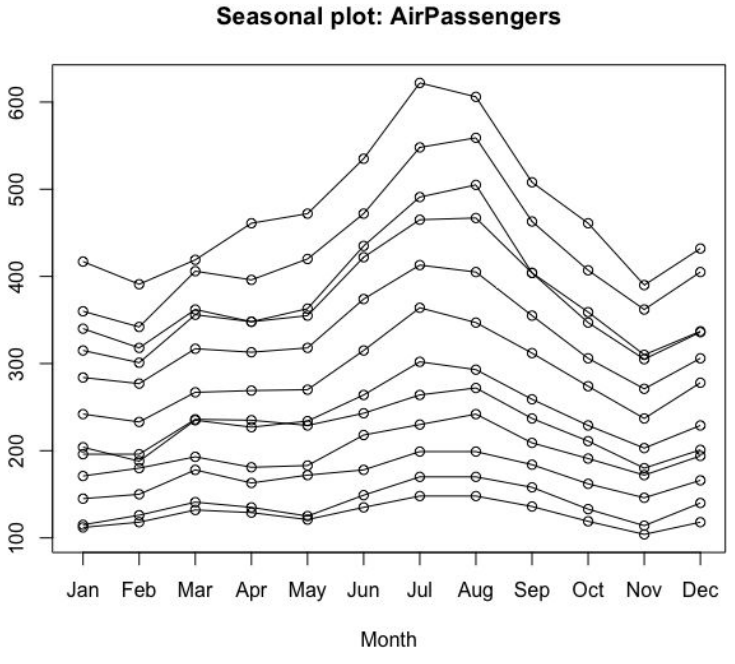

Figure 3-20. We produce a similar seasonal plot more easily with the seasonplot() function.

The different years' curves rarely cross. There was such robust growth that it was rarely the case that different years had the same number of passengers in the same month. There are a few exceptions, but not during the peak months. From these observations alone, we can already produce advice for an air travel company looking to make decisions about how to plan for growth.

An alternate plot of per-month curves against years is less standard but also helpful (see Figure 3-21):

```
## R
> months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
>             "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")

> matplot(t(matrix(AirPassengers, nrow = 12, ncol = 12)),
>             type = 'l', col = colors, lty = 1, lwd = 2.5)
> legend("left", legend = months,
>                         col = colors, lty = 1, lwd = 2.5)
```

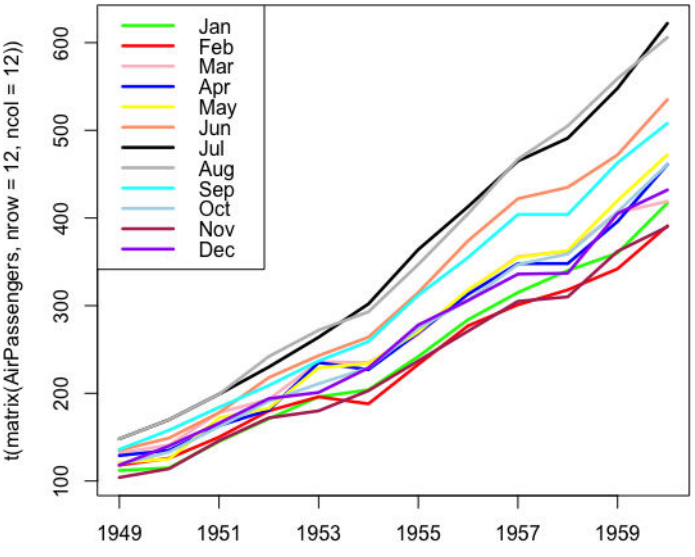

Figure 3-21. Per-month curves of year-to-year time series.[5]

Through the years, the growth trend is accelerating; that is, the rate of increase is itself increasing. Also, two months are growing faster than the others, namely July and August. We can get a similar visualization and similar insights with an easy visualization function supplied by the `forecast` package (see Figure 3-22):

```
## R
> monthplot(AirPassengers)
```

There are two general observations we can make from these plots:

- Time series have more than one useful set of temporal axes against which to plot. We used both an axis of month of the year (January through December) and an axis of years of the data set (the first year up through the last/twelfth year).
- We can glean a lot of useful information and predictive details from visualizations that stack time series data rather than linearly plotting.

5 Visit the GitHub repository to view the original figure, or plot it yourself for a more detailed look.

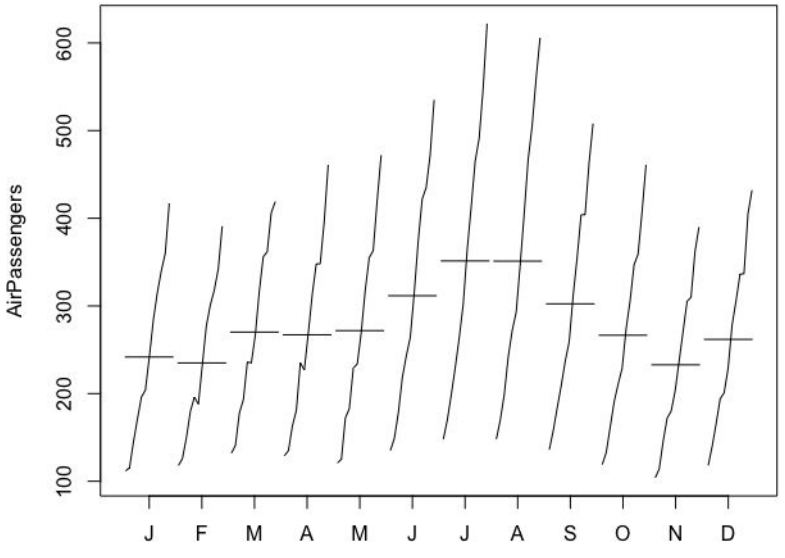

Figure 3-22. By using the monthplot() function, we can see how performance per month changes as the years pass.

We next consider a proper two-dimensional histogram. In a time series context, we can think of a two-dimensional histogram as having one axis for time (or a proxy for time) and another axis for a unit of interest. The "stacked" plots we just did are well on their way to being two-dimensional histograms, but some changes would be helpful:

- We need to bin data both on the time axis and on the number of passengers.
- We need more data. A 2D histogram doesn't make sense until the stacked curves run into one another; they can't properly be seen alone. Otherwise, the 2D histogram fails to convey any additional information.

We generate the 2D histogram on this small data set for illustration and then move on to a more meaningful example. We build our own 2D histogram function from scratch, like so:

```
## R
> hist2d <- function(data, nbins.y, xlabels) {
>   ## we make ybins evenly spaced to include
>   ## minimum and maximum points
>   ymin = min(data)
>   ymax = max(data) * 1.0001
>   ## the lazy way out to avoid worrying about inclusion/exclusion

>   ybins = seq(from = ymin, to = ymax, length.out = nbins.y + 1 )
```

```
>   ## make a zero matrix of the appropriate size
>   hist.matrix = matrix(0, nrow = nbins.y, ncol = ncol(data))

>   ## data comes in matrix form where each row
>   ## represents one data point
>   for (i in 1:nrow(data)) {
>       ts = findInterval(data[i, ], ybins)
>       for (j in 1:ncol(data)) {
>         hist.matrix[ts[j], j] = hist.matrix[ts[j], j] + 1
>       }
>   }
>   hist.matrix
> }
```

We make a histogram with heat map coloring, as follows:

```
## R
> h = hist2d(t(matrix(AirPassengers, nrow = 12, ncol = 12)), 5, months)
> image(1:ncol(h), 1:nrow(h), t(h), col = heat.colors(5),
>       axes = FALSE, xlab = "Time", ylab = "Passenger Count")
```

However, the resulting image (Figure 3-23) is not very satisfying.

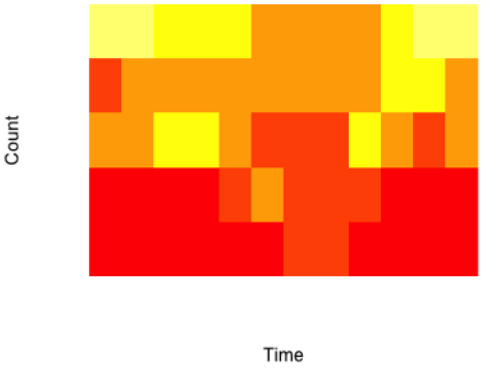

Figure 3-23. Heat map constructed from our homemade 2D histogram of the AirPassengers data.

This plot is not useful because there is not enough data. We only have 12 curves, and we've divided them across 5 buckets. However, an even more important issue is that we don't have stationary data. The use of a histogram assumes a stationary data set. In this case there is a trend, so although we'd like to see seasonality, the trend will necessarily get in the way.

Now we look at a data set that features a larger number of samples and is not polluted by a trend. This is a subset of the FiftyWords data set taken from the UCR Time Series Classification Archive. This data set includes a representation of 50 different words as recorded by a univariate time series, each time series being of equal length.

The subset of the data used to plot Figure 3-24 is from the data set I used in a segment on time series classification in a general tutorial on the subject. You can download that subset, although for the purposes of this exercise you can use either one:

```
## R
> require(data.table)

> words <- fread(url.str)
> w1    <- words[V1 == 1]

> h = hist2d(w1, 25, 1:ncol(w1))

> colors <- gray.colors(20, start = 1, end = .5)
> par(mfrow = c(1, 2))
> image(1:ncol(h), 1:nrow(h), t(h),
>      col = colors, axes = FALSE, xlab = "Time",  ylab = "Projection Value")
> image(1:ncol(h), 1:nrow(h), t(log(h)),
>       col = colors, axes = FALSE, xlab = "Time", ylab = "Projection Value")
```

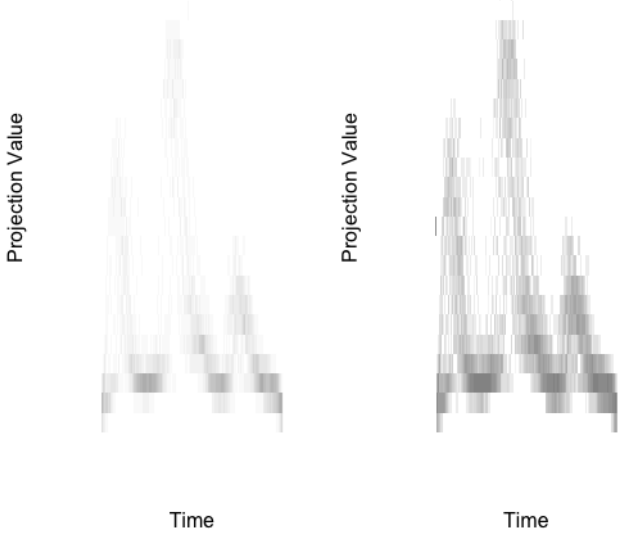

Figure 3-24. Two-dimensional histogram of an audio metric for a single word. The left plot has a linear count scale, while the right plot has a log count scale.

The plot on the right of Figure 3-24 looks better because the counts are colored according to a log transformation of the count rather than the count directly.

This is an application of the same idea as taking the log of a time series to decrease variance and to reduce the importance of, and distance between, outliers. Using a log transformation improves our visualization by not wasting a large portion of the range on the relatively sparse high-count values.

Our homemade option won't be as visually appealing as many precanned options, so we should take a look at these to see what is different. To take advantage of the precanned solutions, we need to reshape our data because these options expect pairs of x-y values to be turned into a 2D histogram. Unlike our homegrown solution, the precanned options for 2D histograms are not designed specifically for time series data. They nonetheless offer excellent visualization solutions (see Figure 3-25):

```
## R
> w1 <- words[V1 == 1]

> ## melt the data to the pairs of paired-coordinates
> ## expected by most 2d histogram implementations
> names(w1) <- c("type", 1:270)
> w1        <- melt(w1, id.vars = "type")

> w1        <- w1[, -1]
> names(w1) <- c("Time point", "Value")

> plot(hexbin(w1))
```

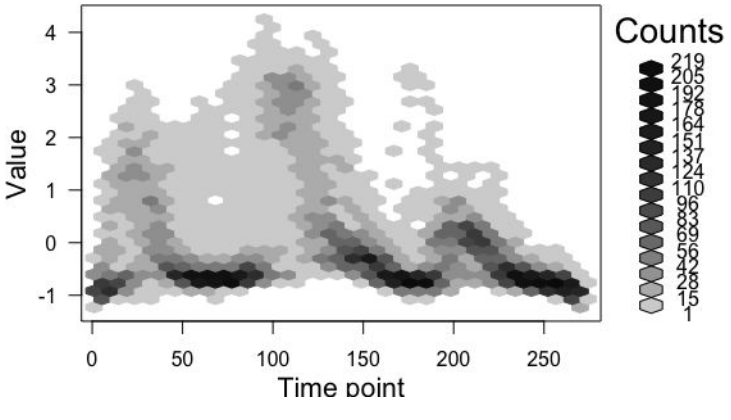

Figure 3-25. An alternative 2D histogram visualization of the same data.

3D Visualizations

3D visualizations do not come with base R, but there are many packages available for them. Here I present some fast plots made with `plotly`, which I chose because it produces plots that can be easily rotated in RStudio and exported to web interfaces. Also, downloading and installing `plotly` tends to be straightforward, which is not true of all visualization packages.

Let's consider the `AirPassengers` data. We plot it in three dimensions, using two dimensions for time (month and year) and one dimension for data values:

```
## R
> require(plotly)
> require(data.table)

> months = 1:12
> ap = data.table(matrix(AirPassengers, nrow = 12, ncol = 12))
> names(ap) = as.character(1949:1960)
> ap[, month := months]
> ap = melt(ap, id.vars = 'month')
> names(ap) = c("month", "year", "count")

> p <- plot_ly(ap, x = ~month, y = ~year, z = ~count,
>              color = ~as.factor(month)) %>%
>   add_markers() %>%
>   layout(scene = list(xaxis = list(title = 'Month'),
>                       yaxis = list(title = 'Year'),
>                       zaxis = list(title = 'PassengerCount')))
```

This 3D visualization helps us get a sense of the overall shape of the data. We have seen much of this before, but expanding to a three-dimensional scatter plot proves to be notably better than a two-dimensional histogram, perhaps because of the paucity of the data (see Figures 3-26 and 3-27).

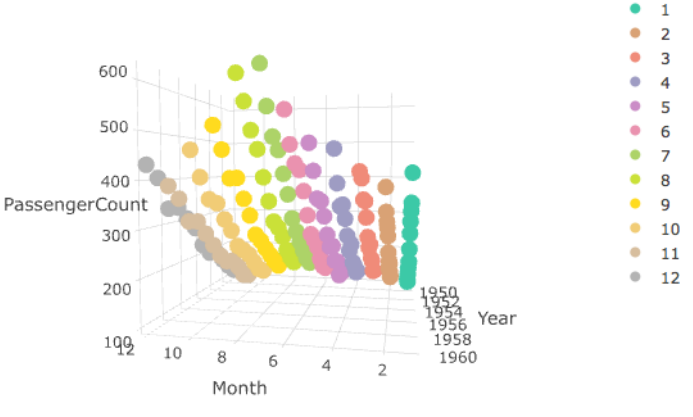

Figure 3-26. A 3D scatterplot of the AirPassenger data. This perspective highlights the seasonality.[6]

6 Visit the GitHub repository to view the original figure, or plot it yourself for a more detailed look.

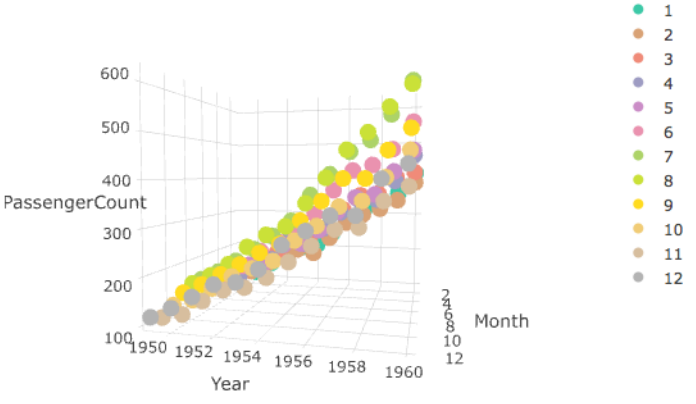

Figure 3-27. Another perspective on the same data more clearly illustrates the increasing trend from one year to the next. I highly recommend running the code on your own computer, where you can rotate it yourself.[7]

We do not necessarily need to expend two axes on time. Instead, we might use two axes for location and one for time. We can visualize a two-dimensional random walk as follows, with a lightly modified example of some `plotly` demonstration code (see Figures 3-28 and 3-29):

```
## R
> file.location <- 'https://raw.githubusercontent.com/plotly/datasets/master/\
                    _3d-line-plot.csv'
> data <- read.csv(file.location)
> p <- plot_ly(data, x = ~x1, y = ~y1, z = ~z1,
>                type = 'scatter3d', mode = 'lines',
>                line = list(color = '#1f77b4', width = 1))
```

The interactive nature of the plot is key. Different perspectives can be misleading or illuminating in ways we can't know until we are able to rotate the data.

7 Visit the GitHub repository to view the original figure, or plot it yourself for a more detailed look.

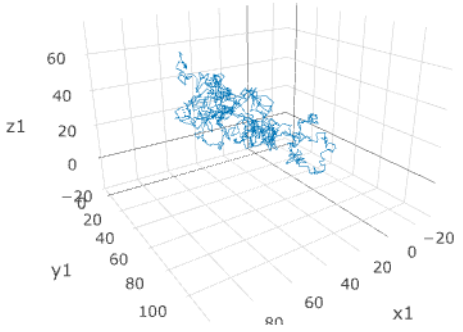

Figure 3-28. One perspective on a two-dimensional random walk over time.

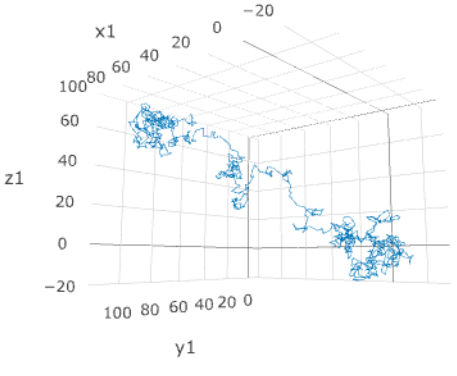

Figure 3-29. This perspective of the same random walk is much more revealing. Again, I encourage you to try this code out for yourself!

A good exercise would be to generate noisy seasonal motion data in two dimensions and visualize this in the same way we have visualized the random walk here. There should be a substantial difference in what you see in that plot compared to the random walk data. Packages like `plotly` can help you experiment quickly and with comprehensive visual feedback.

More Resources

- On spurious correlations:

  Ai Deng, "A Primer on Spurious Statistical Significance in Time Series Regressions," Economics Committee Newsletter 14, no. 1 (2015), https://perma.cc/9CQR-RWHC.

  This industry write-up of what spurious correlations are and how they appear in data is useful for developing some practical insights into when and where to look for this problem in your own data sets. The material is written at a very approachable level.

  Tyler Vigen, Spurious Correlations (New York: Hachette, 2015), https://perma.cc/YY6R-SKWA.

  This collection of ridiculous time series correlations is essential reading for any time series analyst or thinking person.

  Antonio Noriega and Daniel Ventosa-Santaulària, "Spurious Regression Under Broken-Trend Stationarity," Journal of Time Series Analysis 27, no. 5 (2006): 671–84, https://perma.cc/V993-SF4F.

  The authors develop both theory and simulation data to show that shifts in the level or trend of independently and randomly generated data sets influence the presence of spurious correlations.

  C.W.J. Granger and P. Newbold, "Spurious Regressions in Econometrics," Journal of Econometrics 2, no. 2 (1974): 111–20, https://perma.cc/M8TE-AL6U.

  This econometrics article led to a Nobel Prize for identifying the difficulties inherent in dealing with spurious correlations and arguing for a more robust approach to identifying related time series.

- On exploratory data analysis:

  David R. Brillinger and Mark A. Finney, "An Exploratory Data Analysis of the Temperature Fluctuations in a Spreading Fire," Environmetrics 25, no. 6 (2014): 443–53, https://perma.cc/QB3D-APKM.

  This is a very thorough example of how some real-world laboratory data with a geotemporal grid was analyzed by academic and government researchers.

  Robert H. Shumway and David S. Stoffer, "Time Series Regression and Exploratory Data Analysis," in Time Series Analysis and Its Applications with R Examples (New York: Springer, 2011), https://perma.cc/UC5B-TPVS.

  This is a chapter on exploratory data analysis from the authors' canonical treatise on time series analysis for graduate students.

- More visualizations:

  Christian Tominski and Wolfgang Aigner, "The TimeViz Browser," https://perma.cc/94ND-6ZA5.

  This stunning catalog shows examples of, and source code for, many interesting time series visualizations from both academic research papers and industry use cases.

  Oscar Perpiñán Lamigueiro, "GitHub Repository for Displaying Time Series, Spatial, and Space-time Data with R," https://perma.cc/R69Y-5JPL.

  This includes source code for a variety of R-based time series visualizations, including geospatial time series data.

  Myles Harrison, "5 Ways to Do 2D Histograms in R," R-bloggers, September 1, 2014, https://perma.cc/ZCX9-FQQY.

  This is a practical guide to a variety of options offered by R packages to construct 2D histograms and give them meaningful coloring and binning. In addition to a basic overview, a related segment also provides a walkthrough of the `tidyquant` package for visualizing stock market data, an important source of time series.

- On assorted trends:

  Halbert White and Clive W.J. Granger, "Considerations of Trends in Time Series," Journal of Time Series Econometrics 3, no. 1 (2011), https://perma.cc/WF2H-TVTL.

  This recent academic article points out that while trends are ubiquitous in data, even traditional statistics does not have a definitive set of definitions for describing different kinds of trends in data. In an approachable style, the authors offer statistical insights into different ways that nonstationary data may have an underlying trend and guidance on how to improve statistical methods for data.