1 Introduction to basic time series functions in R

This chapter introduces you to some of the basic functions in R for plotting and analyzing univariate time series data. Many of the things you learn here will be relevant when we start examining multivariate time series as well. We will begin with the creation and plotting of time series objects in R, and then moves on to decomposition, differencing, and correlation (*e.g.*, ACF, PACF) before ending with fitting and simulation of ARMA models.

1.1 Time series plots

Time series plots are an excellent way to begin the process of understanding what sort of process might have generated the data of interest. Traditionally, time series have been plotted with the observed data on the *y*-axis and time on the *x*-axis. Sequential time points are usually connected with some form of line, but sometimes other plot forms can be a useful way of conveying important information in the time series (*e.g.*, barplots of sea-surface temperature anomolies show nicely the contrasting El Niño and La Niña phenomena).

Let's start by importing some data; the record of the atmospheric concentration of CO_2 collected at the Mauna Loa Observatory in Hawai'i makes a nice example. The data file contains some extra information that we don't need, so we'll only read in a subset of the columns (*i.e.*, 1, 2 & 5).

```r
## get CO2 data from Mauna Loa observatory
ww1 <- "ftp://aftp.cmdl.noaa.gov/products/"
ww2 <- "trends/co2/co2_mm_mlo.txt"
CO2 <- read.table(text=getURL(paste0(ww1,ww2)))[,c(1,2,5)]
## assign better column names
colnames(CO2) <- c("year","month","ppm")
```

1.1.1 `ts` objects and `plot.ts`

The data are now stored in R as a `data.frame`, but we would like to transform the class to a more user-friendly format for dealing with time series. Fortunately, the `ts` function will do just that, and return an object of class `ts` as well. In addition to the data themselves, we need to provide `ts` with 2 pieces of information about the time index for the data.

The first, `frequency`, is a bit of a misnomer because it does not really refer to the number of cycles per unit time, but rather the number of observations/samples per cycle. So, for example, if the data were collected each hour of a day then `frequency=24`.

The second, `start`, specifies the first sample in terms of (*day*, *hour*), (*year*, *month*), etc. So, for example, if the data were collected monthly beginning in November of 1969, then `frequency=12` and `start=c(1969,11)`. If the data were collected annually, then you simply specify `start` as a scalar (*e.g.*, `start=1991`) and omit `frequency` (*i.e.*, R will set `frequency=1` by default).

The Mauna Loa time series is collected monthly and begins in March of 1958, which we can get from the data themselves, and then pass to `ts`:

```
## create a time series (ts) object from the CO2 data
co2 <- ts(data=CO2$ppm, frequency=12,
          start=c(CO2[1,"year"],CO2[1,"month"]))
```

Now let's plot the data using `plot.ts`, which is designed specifically for `ts` objects like the one we just created above. It's nice because we don't need to specify any *x*-values as they are taken directly from the `ts` object.

```
## plot the ts
plot.ts(co2, ylab=expression(paste("CO"[2]," (ppm)")))
```

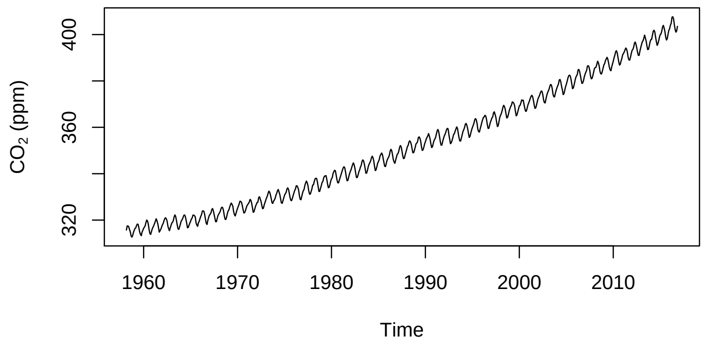

Fig. 1.1. Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i measured monthly from March 1958 to present.

Examination of the plotted time series (Figure 1.1) shows 2 obvious features that would violate any assumption of stationarity: 1) an increasing (and perhaps non-linear) trend over time, and 2) strong seasonal patterns. (*Aside*: Do you know the causes of these 2 phenomena?)

1.1.2 Combining and plotting multiple `ts` objects

Before we examine the CO_2 data further, however, let's see a quick example of how you can combine and plot multiple time series together. We'll begin by getting a second time series (monthly mean temperature anomolies for the Northern Hemisphere) and convert them to a `ts` object.

```
## get N Hemisphere land & ocean temperature anomalies from NOAA
ww1 <- "https://www.ncdc.noaa.gov/cag/time-series/"
ww2 <- "global/nhem/land_ocean/p12/12/1880-2014.csv"
Temp <- read.csv(text=getURL(paste0(ww1,ww2)), skip=3)
## create ts object
tmp <- ts(data=Temp$Value, frequency=12, start=c(1880,1))
```

Before we can plot the two time series together, however, we need to line up their time indices because the temperature data start in January of 1880, but the CO_2 data start in March of 1958. Fortunately, the `ts.intersect` function makes this really easy once the data have been transformed to `ts` objects by trimming the data to a common time frame. Also, `ts.union` works in a similar fashion, but it pads one or both series with the appropriate number of NA's. Let's try both.

```
## intersection (only overlapping times)
datI <- ts.intersect(co2,tmp)
## dimensions of common-time data
dim(datI)
```

```
[1] 682   2
```

```
## union (all times)
datU <- ts.union(co2,tmp)
## dimensions of all-time data
dim(datU)
```

```
[1] 1643    2
```

As you can see, the intersection of the two data sets is much smaller than the union. If you compare them, you will see that the first 938 rows of `datU` contains `NA` in the `co2` column.

It turns out that the regular `plot` function in `R` is smart enough to recognize a `ts` object and use the information contained therein appropriately. Here's how to plot the intersection of the two time series together with the y-axes on alternate sides (results are shown in Figure 1.2):

```
## plot the ts
plot(datI, main="", yax.flip=TRUE)
```

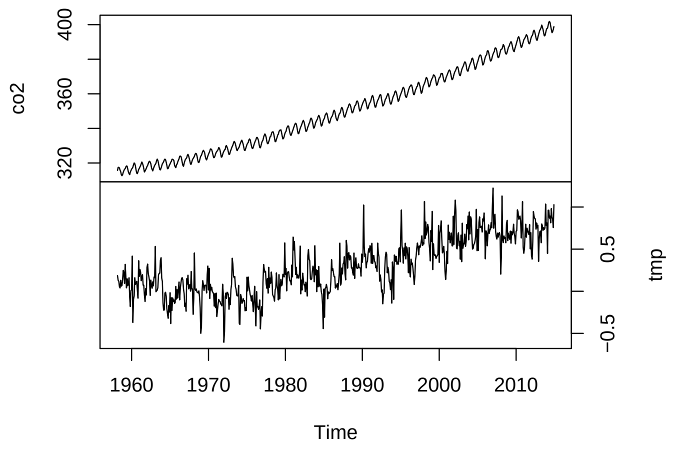

Fig. 1.2. Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) and the mean temperature index for the Northern Hemisphere (bottom) measured monthly from March 1958 to present.

1.2 Decomposition of time series

Plotting time series data is an important first step in analyzing their various components. Beyond that, however, we need a more formal means for identifying and removing characteristics such as a trend or seasonal variation. As discussed in lecture, the decomposition model reduces a time series into 3 components: trend, seasonal effects, and random errors. In turn, we aim to model the random errors as some form of stationary process.

Let's begin with a simple, additive decomposition model for a time series x_t

$$x_t = m_t + s_t + e_t, \tag{1.1}$$

where, at time t, m_t is the trend, s_t is the seasonal effect, and e_t is a random error that we generally assume to have zero-mean and to be correlated over time. Thus, by estimating and subtracting both $\{m_t\}$ and $\{s_t\}$ from $\{x_t\}$, we hope to have a time series of stationary residuals $\{e_t\}$.

1.2.1 Estimating trends

In lecture we discussed how linear filters are a common way to estimate trends in time series. One of the most common linear filters is the moving average, which for time lags from $-a$ to a is defined as

$$\hat{m}_t = \sum_{k=-a}^{a} \left(\frac{1}{1+2a} \right) x_{t+k}. \tag{1.2}$$

This model works well for moving windows of odd-numbered lengths, but should be adjusted for even-numbered lengths by adding only $\frac{1}{2}$ of the 2 most extreme lags so that the filtered value at time t lines up with the original observation at time t. So, for example, in a case with monthly data such as the atmospheric CO_2 concentration where a 12-point moving average would be an obvious choice, the linear filter would be

$$\hat{m}_t = \frac{\frac{1}{2}x_{t-6} + x_{t-5} + \cdots + x_{t-1} + x_t + x_{t+1} + \cdots + x_{t+5} + \frac{1}{2}x_{t+6}}{12} \tag{1.3}$$

It is important to note here that our time series of the estimated trend $\{\hat{m}_t\}$ is actually shorter than the observed time series by $2a$ units.

Conveniently, R has the built-in function `filter` for estimating moving-average (and other) linear filters. In addition to specifying the time series to be filtered, we need to pass in the filter weights (and 2 other arguments we won't worry about here–type `?filter` to get more information). The easiest way to create the filter is with the `rep` function:

```
## weights for moving avg
fltr <- c(1/2,rep(1,times=11),1/2)/12
```

Now let's get our estimate of the trend $\{\hat{m}\}$ with `filter` and plot it:

```
## estimate of trend
co2.trend <- filter(co2, filter=fltr, method="convo", sides=2)
## plot the trend
plot.ts(co2.trend, ylab="Trend", cex=1)
```

The trend is a more-or-less smoothly increasing function over time, the average slope of which does indeed appear to be increasing over time as well (Figure 1.3).

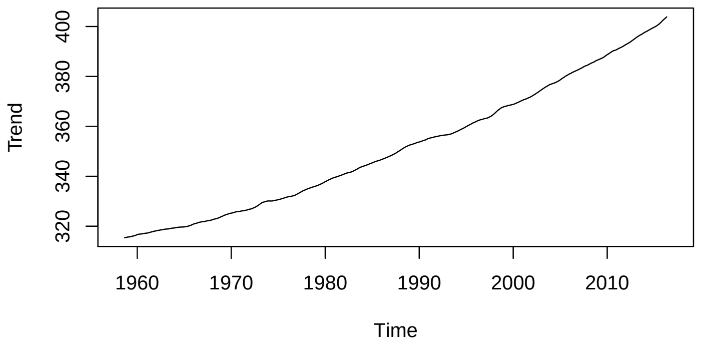

Fig. 1.3. Time series of the estimated trend $\{\hat{m}_t\}$ for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

1.2.2 Estimating seasonal effects

Once we have an estimate of the trend for time t ($\hat{m}_t$) we can easily obtain an estimate of the seasonal effect at time t ($\hat{s}_t$) by subtraction

$$\hat{s}_t = x_t - \hat{m}_t, \tag{1.4}$$

which is really easy to do in R:

```
## seasonal effect over time
co2.1T <- co2 - co2.trend
```

This estimate of the seasonal effect for each time t also contains the random error e_t, however, which can be seen by plotting the time series and careful comparison of Equations (1.1) and (1.4).

```
## plot the monthly seasonal effects
plot.ts(co2.1T, ylab="Seasonal effect", xlab="Month", cex=1)
```

We can obtain the overall seasonal effect by averaging the estimates of $\{\hat{s}_t\}$ for each month and repeating this sequence over all years.

```
## length of ts
ll <- length(co2.1T)
## frequency (ie, 12)
ff <- frequency(co2.1T)
## number of periods (years); %/% is integer division
periods <- ll %/% ff
## index of cumulative month
index <- seq(1,ll,by=ff) - 1
## get mean by month
mm <- numeric(ff)
```

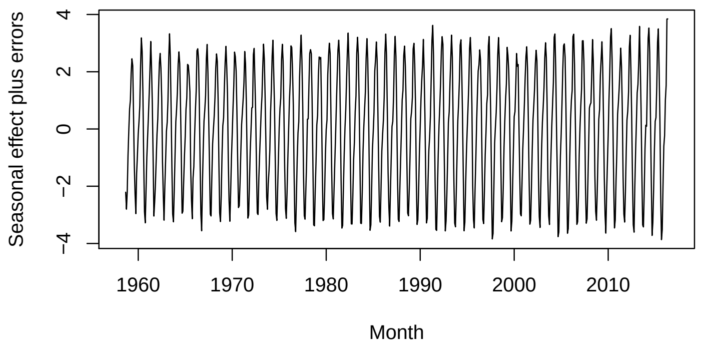

Fig. 1.4. Time series of seasonal effects plus random errors for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i, measured monthly from March 1958 to present.

```
for(i in 1:ff) {
  mm[i] <- mean(co2.1T[index+i], na.rm=TRUE)
}
## subtract mean to make overall mean=0
mm <- mm - mean(mm)
```

Before we create the entire time series of seasonal effects, let's plot them for each month to see what is happening within a year:

```
## plot the monthly seasonal effects
plot.ts(mm, ylab="Seasonal effect", xlab="Month", cex=1)
```

It looks like, on average, that the CO_2 concentration is highest in spring (March) and lowest in summer (August) (Figure 1.5). (*Aside*: Do you know why this is?)

Finally, let's create the entire time series of seasonal effects $\{\hat{s}_t\}$:

```
## create ts object for season
co2.seas <- ts(rep(mm, periods+1)[seq(ll)],
               start=start(co2.1T),
               frequency=ff)
```

1.2.3 Completing the model

The last step in completing our full decomposition model is obtaining the random errors $\{\hat{e}_t\}$, which we can get via simple subtraction

$$\hat{e}_t = x_t - \hat{m}_t - \hat{s}_t. \tag{1.5}$$

Again, this is really easy in `R`:

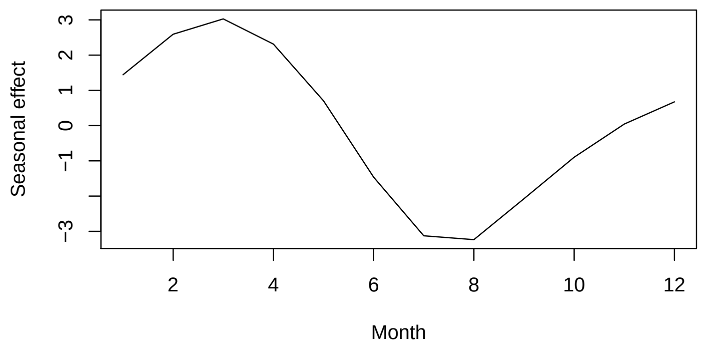

Fig. 1.5. Estimated monthly seasonal effects for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

```
## random errors over time
co2.err <- co2 - co2.trend - co2.seas
```

Now that we have all 3 of our model components, let's plot them together with the observed data $\{x_t\}$. The results are shown in Figure 1.6.

```
## plot the obs ts, trend & seasonal effect
plot(cbind(co2,co2.trend,co2.seas,co2.err),main="",yax.flip=TRUE)
```

1.2.4 Using `decompose` for decomposition

Now that we have seen how to estimate and plot the various components of a classical decomposition model in a piecewise manner, let's see how to do this in one step in `R` with the function `decompose`, which accepts a `ts` object as input and returns an object of class `decomposed.ts`.

```
## decomposition of CO2 data
co2.decomp <- decompose(co2)
```

`co2.decomp` is a list with the following elements, which should be familiar by now:

`x`	the observed time series $\{x_t\}$
`seasonal`	time series of estimated seasonal component $\{\hat{s}_t\}$
`figure`	mean seasonal effect (`length(figure) == frequency(x)`)
`trend`	time series of estimated trend $\{\hat{m}_t\}$
`random`	time series of random errors $\{\hat{e}_t\}$
`type`	type of error (`"additive"` or `"multiplicative"`)

We can easily make plots of the output and compare them to those in Figure 1.6:

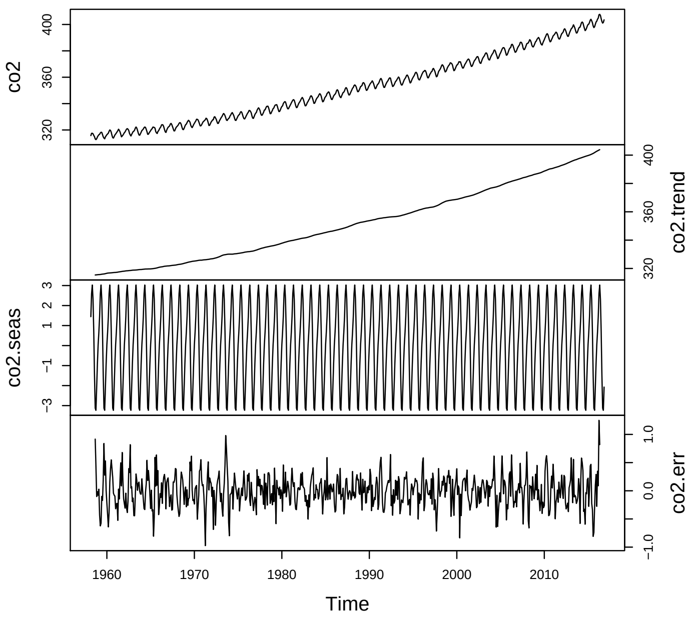

Fig. 1.6. Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors.

```
## plot the obs ts, trend & seasonal effect
plot(co2.decomp, yax.flip=TRUE)
```

The results obtained with `decompose` (Figure 1.7) are identical to those we estimated previously.

Another nice feature of the `decompose` function is that it can be used for decomposition models with multiplicative (*i.e.*, non-additive) errors (*e.g.*, if the original time series had a seasonal amplitude that increased with time). To do, so pass in the argument `type="multiplicative"`, which is set to `type="additive"` by default.

1.3 Differencing to remove a trend or seasonal effects

An alternative to decomposition for removing trends is differencing. We saw in lecture how the difference operator works and how it can be used to remove linear and nonlinear trends as well as various seasonal features that might be evident in the data. As a reminder, we define the difference operator as

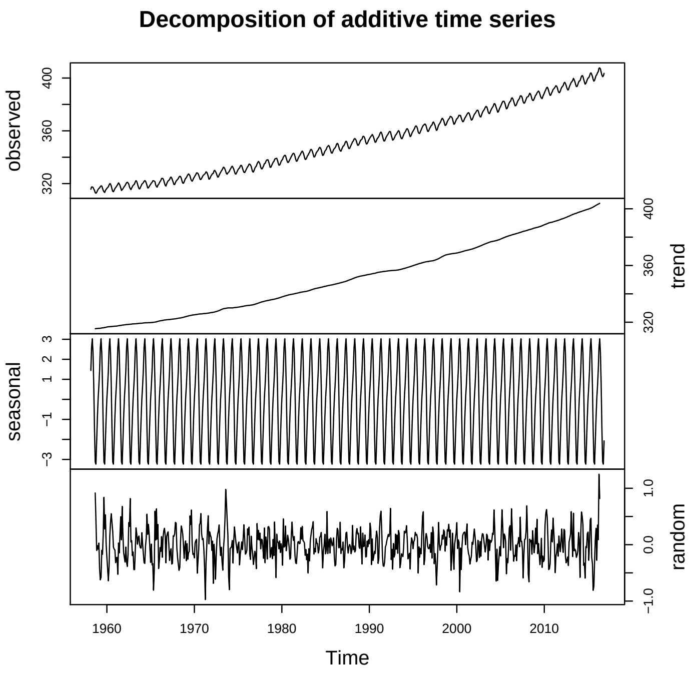

Fig. 1.7. Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors obtained with the function `decompose`.

$$\nabla x_t = x_t - x_{t-1}, \tag{1.6}$$

and, more generally, for order d

$$\nabla^d x_t = (1 - \mathbf{B})^d x_t, \tag{1.7}$$

where $\mathbf{B}$ is the backshift operator (*i.e.*, $\mathbf{B}^k x_t = x_{t-k}$ for $k \geq 1$).

So, for example, a random walk is one of the most simple and widely used time series models, but it is not stationary. We can write a random walk model as

$$x_t = x_{t-1} + w_t, \text{ with } w_t \sim \text{N}(0, q). \tag{1.8}$$

Applying the difference operator to Equation (1.8) will yield a time series of Gaussian white noise errors $\{w_t\}$:

$$\begin{aligned} \nabla(x_t &= x_{t-1} + w_t) \\ x_t - x_{t-1} &= x_{t-1} - x_{t-1} + w_t \\ x_t - x_{t-1} &= w_t \end{aligned} \tag{1.9}$$

1.3.1 Using the `diff` function

In `R` we can use the `diff` function for differencing a time series, which requires 3 arguments: `x` (the data), `lag` (the lag at which to difference), and `differences` (the order of differencing; d in Equation (1.7)). For example, first-differencing a time series will remove a linear trend (*i.e.*, `differences=1`); twice-differencing will remove a quadratic trend (*i.e.*, `differences=2`). In addition, first-differencing a time series at a lag equal to the period will remove a seasonal trend (*e.g.*, set `lag=12` for monthly data).

Let's use `diff` to remove the trend and seasonal signal from the CO_2 time series, beginning with the trend. Close inspection of Figure 1.1 would suggest that there is a nonlinear increase in CO_2 concentration over time, so we'll set `differences=2`):

```
## twice-difference the CO2 data
co2.D2 <- diff(co2, differences=2)
## plot the differenced data
plot(co2.D2, ylab=expression(paste(nabla^2,"CO"[2])))
```

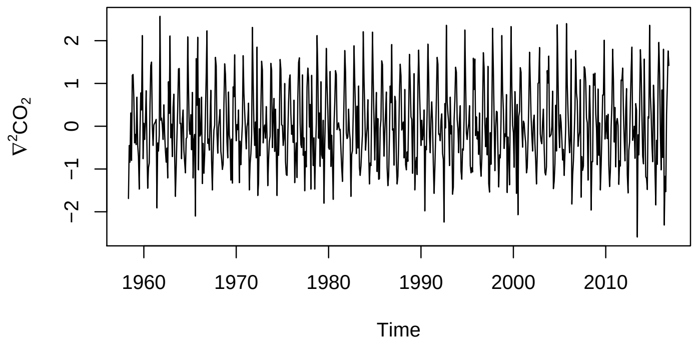

Fig. 1.8. Time series of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

We were apparently successful in removing the trend, but the seasonal effect still appears obvious (Figure 1.8). Therefore, let's go ahead and difference that series at lag-12 because our data were collected monthly.

```
## difference the differenced CO2 data
co2.D2D12 <- diff(co2.D2, lag=12)
## plot the newly differenced data
plot(co2.D2D12,
     ylab=expression(paste(nabla,"(",nabla^2,"CO"[2],")")))
```

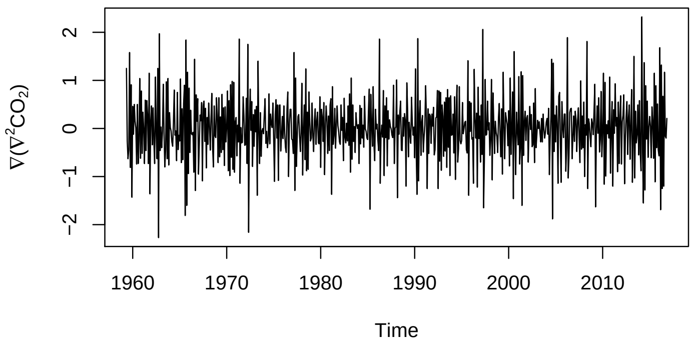

Fig. 1.9. Time series of the lag-12 difference of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

Now we have a time series that appears to be random errors without any obvious trend or seasonal components (Figure 1.9).

1.4 Correlation within and among time series

The concepts of covariance and correlation are very important in time series analysis. In particular, we can examine the correlation structure of the original data or random errors from a decomposition model to help us identify possible form(s) of (non)stationary model(s) for the stochastic process.

1.4.1 Autocorrelation function (ACF)

Autocorrelation is the correlation of a variable with itself at differing time lags. Recall from lecture that we defined the sample autocovariance function (ACVF), c_k, for some lag k as

$$c_k = \frac{1}{n} \sum_{t=1}^{n-k} (x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{1.10}$$

Note that the sample autocovariance of $\{x_t\}$ at lag 0, c_0, equals the sample variance of $\{x_t\}$ calculated with a denominator of n. The sample autocorrelation function (ACF) is defined as

$$r_k = \frac{c_k}{c_0} = \mathrm{Cor}(x_t, x_{t+k}) \tag{1.11}$$

Recall also that an approximate 95% confidence interval on the ACF can be estimated by

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{1.12}$$

where n is the number of data points used in the calculation of the ACF.

It is important to remember two things here. First, although the confidence interval is commonly plotted and interpreted as a horizontal line over all time lags, the interval itself actually grows as the lag increases because the number of data points n used to estimate the correlation decreases by 1 for every integer increase in lag. Second, care must be exercised when interpreting the "significance" of the correlation at various lags because we should expect, *a priori*, that approximately 1 out of every 20 correlations will be significant based on chance alone.

We can use the `acf` function in `R` to compute the sample ACF (note that adding the option `type="covariance"` will return the sample auto-covariance (ACVF) instead of the ACF–type `?acf` for details). Calling the function by itself will will automatically produce a correlogram (*i.e.*, a plot of the autocorrelation versus time lag). The argument `lag.max` allows you to set the number of positive and negative lags. Let's try it for the CO_2 data.

```
## correlogram of the CO2 data
acf(co2, lag.max=36)
```

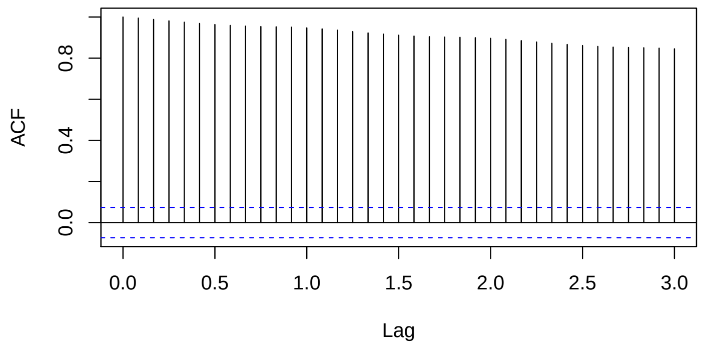

Fig. 1.10. Correlogram of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `acf`.

There are 4 things about Figure 1.10 that are noteworthy:

1. the ACF at lag 0, r_0, equals 1 by default (*i.e.*, the correlation of a time series with itself)–it's plotted as a reference point;
2. the x-axis has decimal values for lags, which is caused by `R` using the year index as the lag rather than the month;
3. the horizontal blue lines are the approximate 95% CI's; and

4. there is very high autocorrelation even out to lags of 36 months.

As an alternative to the plotting utility in `acf`, let's define a new plot function for `acf` objects with some better features:

```
plot.acf <- function(ACFobj) {
  rr <- ACFobj$acf[-1]
  kk <- length(rr)
  nn <- ACFobj$n.used
  plot(seq(kk),rr,type="h",lwd=2,yaxs="i",xaxs="i",
       ylim=c(floor(min(rr)),1),xlim=c(0,kk+1),
       xlab="Lag",ylab="Correlation",las=1)
  abline(h=-1/nn+c(-2,2)/sqrt(nn),lty="dashed",col="blue")
  abline(h=0)
}
```

Now we can assign the result of `acf` to a variable and then use the information contained therein to plot the correlogram with our new plot function.

```
## acf of the CO2 data
co2.acf <- acf(co2, lag.max=36)
## correlogram of the CO2 data
plot.acf(co2.acf)
```

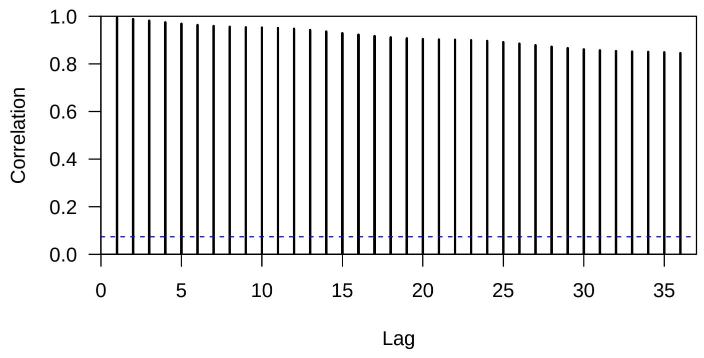

Fig. 1.11. Correlogram of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `plot.acf`.

Notice that all of the relevant information is still there (Figure 1.11), but now $r_0 = 1$ is not plotted at lag-0 and the lags on the x-axis are displayed correctly as integers.

Before we move on to the PACF, let's look at the ACF for some deterministic time series, which will help you identify interesting properties (*e.g.*, trends, seasonal effects) in a stochastic time series, and account for them in time series models–an important topic in this course. First, let's look at a straight line.

```
## length of ts
nn <- 100
## create straight line
tt <- seq(nn)
## set up plot area
par(mfrow=c(1,2))
## plot line
plot.ts(tt, ylab=expression(italic(x[t])))
## get ACF
line.acf <- acf(tt, plot=FALSE)
## plot ACF
plot.acf(line.acf)
```

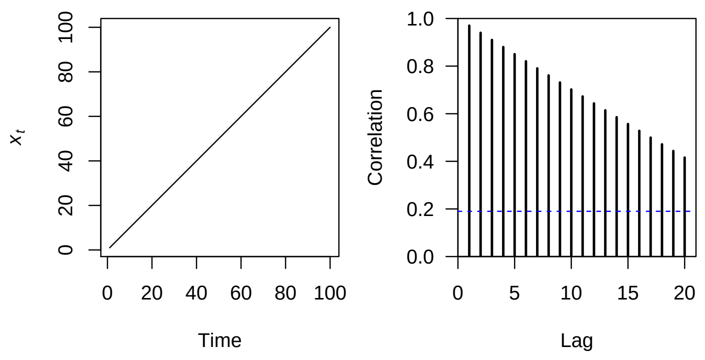

Fig. 1.12. Time series plot of a straight line (left) and the correlogram of its ACF (right).

The correlogram for a straight line is itself a linearly decreasing function over time (Figure 1.12).

Now let's examine the ACF for a sine wave and see what sort of pattern arises.

```
## create sine wave
tt <- sin(2*pi*seq(nn)/12)
## set up plot area
par(mfrow=c(1,2))
## plot line
plot.ts(tt, ylab=expression(italic(x[t])))
```

```
## get ACF
sine.acf <- acf(tt, plot=FALSE)
## plot ACF
plot.acf(sine.acf)
```

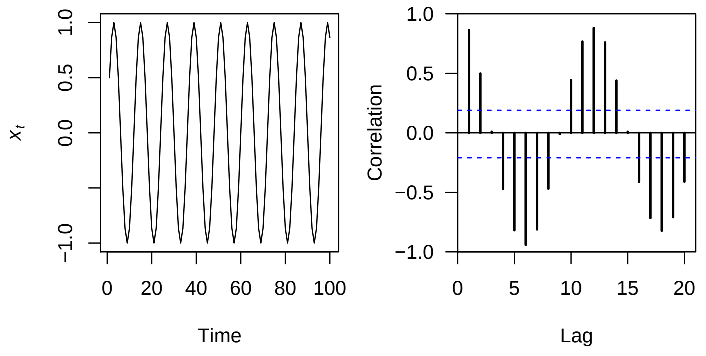

Fig. 1.13. Time series plot of a discrete sine wave (left) and the correlogram of its ACF (right).

Perhaps not surprisingly, the correlogram for a sine wave is itself a sine wave whose amplitude decreases linearly over time (Figure 1.13).

Now let's examine the ACF for a sine wave with a linear downward trend and see what sort of patterns arise.

```
## create sine wave with trend
tt <- sin(2*pi*seq(nn)/12) - seq(nn)/50
## set up plot area
par(mfrow=c(1,2))
## plot line
plot.ts(tt, ylab=expression(italic(x[t])))
## get ACF
sili.acf <- acf(tt, plot=FALSE)
## plot ACF
plot.acf(sili.acf)
```

The correlogram for a sine wave with a trend is itself a nonsymmetrical sine wave whose amplitude and center decrease over time (Figure 1.14).

As we have seen, the ACF is a powerful tool in time series analysis for identifying important features in the data. As we will see later, the ACF is also an important diagnostic tool for helping to select the proper order of p and q in ARMA(p,q) models.

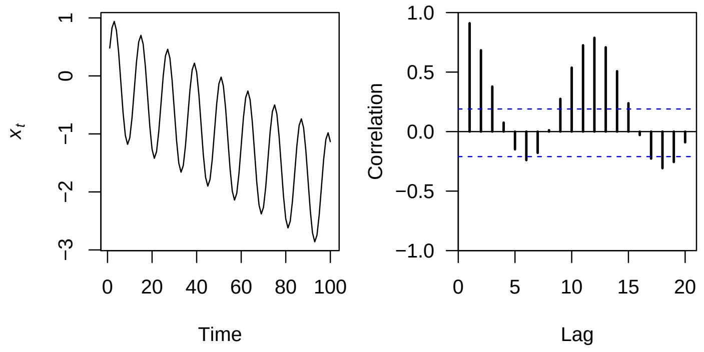

Fig. 1.14. Time series plot of a discrete sine wave (left) and the correlogram of its ACF (right).

1.4.2 Partial autocorrelation function (PACF)

The partial autocorrelation function (PACF) measures the linear correlation of a series $\{x_t\}$ and a lagged version of itself $\{x_{t+k}\}$ with the linear dependence of $\{x_{t-1}, x_{t-2}, \dots, x_{t-(k-1)}\}$ removed. Recall from lecture that we define the PACF as

$$f_k = \begin{cases} \text{Cor}(x_1, x_0) = r_1 & \text{if } k = 1; \\ \text{Cor}(x_k - x_k^{k-1}, x_0 - x_0^{k-1}) & \text{if } k \geq 2; \end{cases} \tag{1.13}$$

with

$$x_k^{k-1} = \beta_1 x_{k-1} + \beta_2 x_{k-2} + \dots + \beta_{k-1} x_1; \tag{1.14a}$$

$$x_0^{k-1} = \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_{k-1} x_{k-1}. \tag{1.14b}$$

It's easy to compute the PACF for a variable in `R` using the `pacf` function, which will automatically plot a correlogram when called by itself (similar to `acf`). Let's look at the PACF for the CO_2 data.

```
## PACF of the CO2 data
pacf(co2, lag.max=36)
```

The default plot for PACF is a bit better than for ACF, but here is another plotting function that might be useful.

```
plot.pacf <- function(PACFobj) {
  rr <- PACFobj$acf
  kk <- length(rr)
  nn <- PACFobj$n.used
  plot(seq(kk),rr,type="h",lwd=2,yaxs="i",xaxs="i",
       ylim=c(floor(min(rr)),1),xlim=c(0,kk+1),
       xlab="Lag",ylab="PACF",las=1)
  abline(h=-1/nn+c(-2,2)/sqrt(nn),lty="dashed",col="blue")
  abline(h=0)
}
```

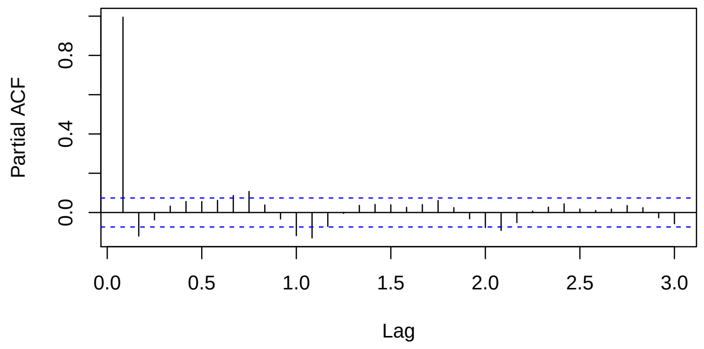

Fig. 1.15. Correlogram of the PACF for the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `pacf`.

Notice in Figure 1.15 that the partial autocorrelation at lag-1 is very high (it equals the ACF at lag-1), but the other values at lags > 1 are relatively small, unlike what we saw for the ACF. We will discuss this in more detail later on in this lab.

Notice also that the PACF plot again has real-valued indices for the time lag, but it does not include any value for lag-0 because it is impossible to remove any intermediate autocorrelation between t and $t-k$ when $k=0$, and therefore the PACF does not exist at lag-0. If you would like, you can use the `plot.acf` function we defined above to plot the PACF estimates because `acf` and `pacf` produce identical list structures (results not shown here).

```
## PACF of the CO2 data
co2.pacf <- pacf(co2)
## correlogram of the CO2 data
plot.acf(co2.pacf)
```

As with the ACF, we will see later on how the PACF can also be used to help identify the appropriate order of p and q in ARMA(p,q) models.

1.4.3 Cross-correlation function (CCF)

Often we are interested in looking for relationships between 2 different time series. There are many ways to do this, but a simple method is via examination of their cross-covariance and cross-correlation.

We begin by defining the sample cross-covariance function (CCVF) in a manner similar to the ACVF, in that

$$g_k^{xy} = \frac{1}{n}\sum_{t=1}^{n-k}(y_t - \bar{y})(x_{t+k} - \bar{x}), \tag{1.15}$$

but now we are estimating the correlation between a variable y and a *different* time-shifted variable x_{t+k}. The sample cross-correlation function (CCF) is then defined analogously to the ACF, such that

$$r_k^{xy} = \frac{g_k^{xy}}{\sqrt{\mathrm{SD}_x \mathrm{SD}_y}}; \tag{1.16}$$

SD_x and SD_y are the sample standard deviations of $\{x_t\}$ and $\{y_t\}$, respectively. It is important to re-iterate here that $r_k^{xy} \neq r_{-k}^{xy}$, but $r_k^{xy} = r_{-k}^{yx}$. Therefore, it is very important to pay particular attention to which variable you call y (*i.e.*, the "response") and which you call x (*i.e.*, the "predictor").

As with the ACF, an approximate 95% confidence interval on the CCF can be estimated by

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{1.17}$$

where n is the number of data points used in the calculation of the CCF, and the same assumptions apply to its interpretation.

Computing the CCF in `R` is easy with the function `ccf` and it works just like `acf`. In fact, `ccf` is just a "wrapper" function that calls `acf`. As an example, let's examine the CCF between sunspot activity and number of lynx trapped in Canada as in the classic paper by Moran[1].

To begin, let's get the data, which are conveniently included in the `datasets` package included as part of the base installation of `R`. Before calculating the CCF, however, we need to find the matching years of data. Again, we'll use the `ts.intersect` function.

```
## get the matching years of sunspot data
suns <- ts.intersect(lynx,sunspot.year)[,"sunspot.year"]
## get the matching lynx data
lynx <- ts.intersect(lynx,sunspot.year)[,"lynx"]
```

Here are plots of the time series.

```
## plot time series
plot(cbind(suns,lynx), yax.flip=TRUE)
```

It is important to remember which of the 2 variables you call y and x when calling `ccf(x, y, ...)`. In this case, it seems most relevant to treat

[1] Moran, P.A.P. 1949. The statistical analysis of the sunspot and lynx cycles. *J. Anim. Ecol.* 18:115-116

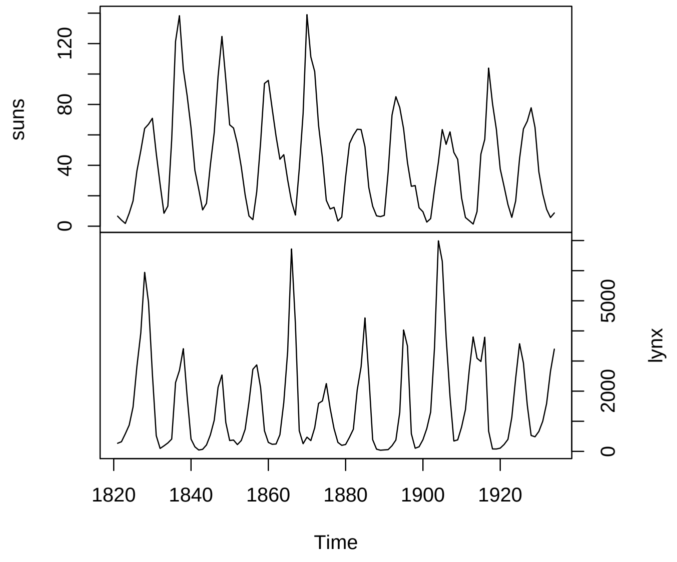

Fig. 1.16. Time series of sunspot activity (top) and lynx trappings in Canada (bottom) from 1821-1934.

lynx as the y and sunspots as the x, in which case we are mostly interested in the CCF at negative lags (*i.e.*, when sunspot activity predates inferred lynx abundance). Furthermore, we'll use log-transformed lynx trappings.

```
## CCF of sunspots and lynx
ccf(suns, log(lynx), ylab="Cross-correlation")
```

From Figures 1.16 and 1.17 it looks like lynx numbers are relatively low 3-5 years after high sunspot activity (*i.e.*, significant correlation at lags of -3 to -5).

1.5 White noise (WN)

A time series $\{w_t\}$ is a discrete white noise series (DWN) if the $w_1, w_1, \dots, w_t$ are independent and identically distributed (IID) with a mean of zero. For most of the examples in this course we will assume that the $w_t \sim \mathrm{N}(0, q)$, and therefore we refer to the time series $\{w_t\}$ as Gaussian white noise. If our time series

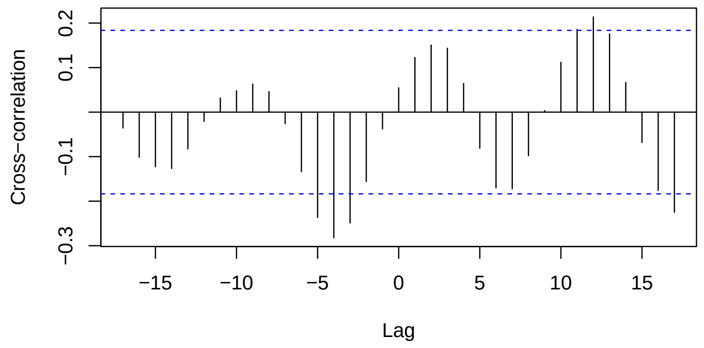

Fig. 1.17. CCF for annual sunspot activity and the log of the number of lynx trappings in Canada from 1821-1934.

model has done an adequate job of removing all of the serial autocorrelation in the time series with trends, seasonal effects, etc., then the model residuals ($e_t = y_t - \hat{y}_t$) will be a WN sequence with the following properties for its mean ($\bar{e}$), covariance (c_k), and autocorrelation (r_k):

$$\begin{aligned}
\bar{x} &= 0 \\
c_k &= \text{Cov}(e_t, e_{t+k}) = \begin{cases} q & \text{if } k = 0 \\ 0 & \text{if } k \neq 1 \end{cases} \\
r_k &= \text{Cor}(e_t, e_{t+k}) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k \neq 1. \end{cases}
\end{aligned} \tag{1.18}$$

1.5.1 Simulating white noise

Simulating WN in `R` is straightforward with a variety of built-in random number generators for continuous and discrete distributions. Once you know `R`'s abbreviation for the distribution of interest, you add an `r` to the beginning to get the function's name. For example, a Gaussian (or normal) distribution is abbreviated `norm` and so the function is `rnorm`. All of the random number functions require two things: the number of samples from the distribution (`n`), and the parameters for the distribution itself (*e.g.*, mean & SD of a normal). Check the help file for the distribution of interest to find out what parameters you must specify (*e.g.*, type `?rnorm` to see the help for a normal distribution).

Here's how to generate 100 samples from a normal distribution with mean of 5 and standard deviation of 0.2, and 50 samples from a Poisson distribution with a rate (λ) of 20.

```
set.seed(123)
## random normal variates
```

```
GWN <- rnorm(n=100, mean=5, sd=0.2)
## random Poisson variates
PWN <- rpois(n=50, lambda=20)
```

Here are plots of the time series. Notice that on one occasion the same number was drawn twice in a row from the Poisson distribution, which is discrete. That is virtually guaranteed to never happen with a continuous distribution.

```
## set up plot region
par(mfrow=c(1,2))
## plot normal variates with mean
plot.ts(GWN)
abline(h=5, col="blue", lty="dashed")
## plot Poisson variates with mean
plot.ts(PWN)
abline(h=20, col="blue", lty="dashed")
```

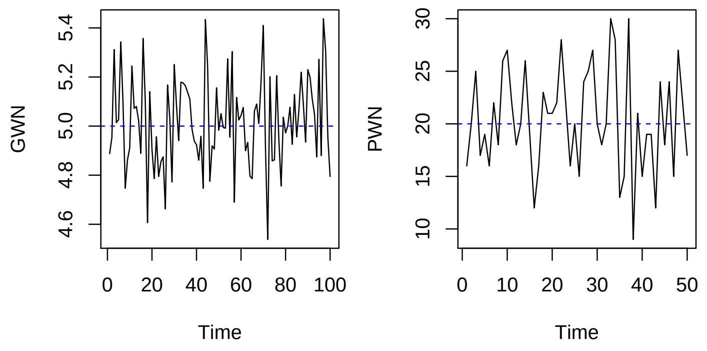

Fig. 1.18. Time series plots of simulated Gaussian (left) and Poisson (right) white noise.

Now let's examine the ACF for the 2 white noise series and see if there is, in fact, zero autocorrelation for lags ≥ 1.

```
## set up plot region
par(mfrow=c(1,2))
## plot normal variates with mean
acf(GWN, main="", lag.max=20)
## plot Poisson variates with mean
acf(PWN, main="", lag.max=20)
```

Interestingly, the r_k are all greater than zero in absolute value although they are not statistically different from zero for lags 1-20. This is because we

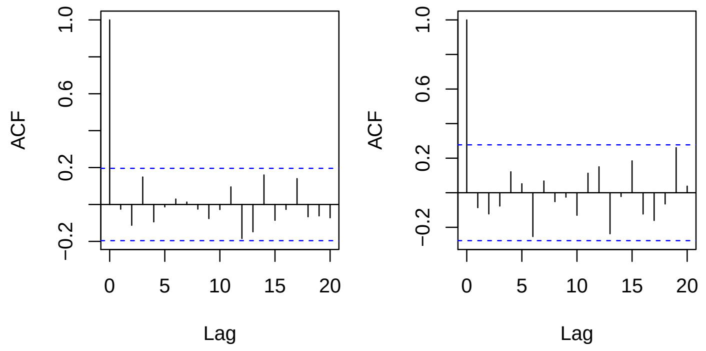

Fig. 1.19. ACF's for the simulated Gaussian (left) and Poisson (right) white noise shown in Figure 1.18.

are dealing with a *sample* of the distributions rather than the entire population of all random variates. As an exercise, try setting `n=1e6` instead of `n=100` or `n=50` in the calls calls above to generate the WN sequences and see what effect it has on the estimation of r_k. It is also important to remember, as we discussed earlier, that we should expect that approximately 1 in 20 of the r_k will be statistically greater than zero based on chance alone, especially for relatively small sample sizes, so don't get too excited if you ever come across a case like then when inspecting model residuals.

1.6 Random walks (RW)

Random walks receive considerable attention in time series analyses because of their ability to fit a wide range of data despite their surprising simplicity. In fact, random walks are the most simple non-stationary time series model. A random walk is a time series $\{x_t\}$ where

$$x_t = x_{t-1} + w_t, \tag{1.19}$$

and w_t is a discrete white noise series where all values are independent and identically distributed (IID) with a mean of zero. In practice, we will almost always assume that the w_t are Gaussian white noise, such that $w_t \sim \mathrm{N}(0,q)$. We will see later that a random walk is a special case of an autoregressive model.

1.6.1 Simulating a random walk

Simulating a RW model in `R` is straightforward with a `for` loop and the use of `rnorm` to generate Gaussian errors (type `?rnorm` to see details on the function and its useful relatives `dnorm` and `pnorm`). Let's create 100 obs (we'll also set the random number seed so everyone gets the same results).

```
## set random number seed
set.seed(123)
## length of time series
TT <- 100
## initialize {x_t} and {w_t}
xx <- ww <- rnorm(n=TT, mean=0, sd=1)
## compute values 2 thru TT
for(t in 2:TT) { xx[t] <- xx[t-1] + ww[t] }
```

Now let's plot the simulated time series and its ACF.

```
## setup plot area
par(mfrow=c(1,2))
## plot line
plot.ts(xx, ylab=expression(italic(x[t])))
## plot ACF
plot.acf(acf(xx, plot=FALSE))
```

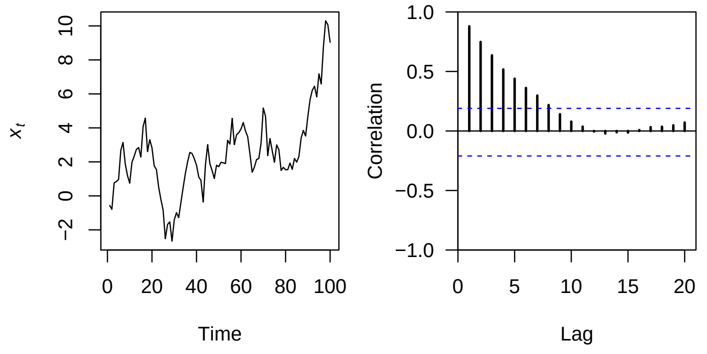

Fig. 1.20. Simulated time series of a random walk model (left) and its associated ACF (right).

Perhaps not surprisingly based on their names, autoregressive models such as RW's have a high degree of autocorrelation out to long lags (Figure 1.20).

1.6.2 Alternative formulation of a random walk

As an aside, let's use an alternative formulation of a random walk model to see an even shorter way to simulate an RW in R. Based on our definition of a random walk in Equation (1.19), it is easy to see that

$$\begin{aligned}x_t &= x_{t-1} + w_t \\ x_{t-1} &= x_{t-2} + w_{t-1} \\ x_{t-2} &= x_{t-3} + w_{t-2} \\ &\vdots\end{aligned} \tag{1.20}$$

Therefore, if we substitute $x_{t-2} + w_{t-1}$ for x_{t-1} in the first equation, and then $x_{t-3} + w_{t-2}$ for x_{t-2}, and so on in a recursive manner, we get

$$x_t = w_t + w_{t-1} + w_{t-2} + \cdots + w_{t-\infty} + x_{t-\infty}. \tag{1.21}$$

In practice, however, the time series will not start an infinite time ago, but rather at some $t = 1$, in which case we can write

$$\begin{aligned}x_t &= w_1 + w_2 + \cdots + w_t \\ &= \sum_{t=1}^{T} w_t.\end{aligned} \tag{1.22}$$

From Equation (1.22) it is easy to see that the value of an RW process at time step t is the sum of all the random errors up through time t. Therefore, in `R` we can easily simulate a realization from an RW process using the `cumsum(x)` function, which does cumulative summation of the vector `x` over its entire length. If we use the same errors as before, we should get the same results.

```
## simulate RW
x2 <- cumsum(ww)
```

Let's plot both time series to see if it worked.

```
## setup plot area
par(mfrow=c(1,2))
## plot 1st RW
plot.ts(xx, ylab=expression(italic(x[t])))
## plot 2nd RW
plot.ts(x2, ylab=expression(italic(x[t])))
```

Indeed, both methods of generating a RW time series appear to be equivalent.

1.7 Autoregressive (AR) models

Autoregressive models of order p, abbreviated AR(p), are commonly used in time series analyses. In particular, AR(1) models (and their multivariate extensions) see considerable use in ecology as we will see later in the course. Recall from lecture that an AR(p) model is written as

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t, \tag{1.23}$$

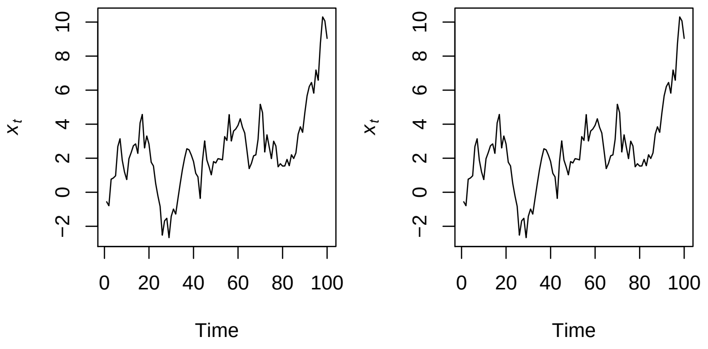

Fig. 1.21. Time series of the same random walk model formulated as Equation (1.19) and simulated via a `for` loop (left), and as Equation (1.22) and simulated via `cumsum` (right).

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2. For our purposes we usually assume that $w_t \sim \text{N}(0,q)$. Note that the random walk in Equation (1.19) is a special case of an AR(1) model where $\phi_1 = 1$ and $\phi_k = 0$ for $k \geq 2$.

1.7.1 Simulating an AR(p) process

Although we could simulate an AR(p) process in `R` using a `for` loop just as we did for a random walk, it's much easier with the function `arima.sim`, which works for all forms and subsets of ARIMA models. To do so, remember that the AR in ARIMA stands for "autoregressive", the I for "integrated", and the MA for "moving-average"; we specify the order of ARIMA models as p,d,q. So, for example, we would specify an AR(2) model as ARIMA(2,0,0), or an MA(1) model as ARIMA(0,0,1). If we had an ARMA(3,1) model that we applied to data that had been twice-differenced, then we would have an ARIMA(3,2,1) model.

`arima.sim` will accept many arguments, but we are interested primarily in two of them: `n` and `model` (type `?arima.sim` to learn more). The former simply indicates the length of desired time series, but the latter is more complex. Specifically, `model` is a list with the following elements:

`order`	a vector of length 3 containing the ARIMA(p,d,q) order
`ar`	a vector of length p containing the AR(p) coefficients
`ma`	a vector of length q containing the MA(q) coefficients
`sd`	a scalar indicating the std dev of the Gaussian errors

Note that you can omit the `ma` element entirely if you have an AR(p) model, or omit the `ar` element if you have an MA(q) model. If you omit the `sd` element, `arima.sim` will assume you want normally distributed errors with SD = 1.

Also note that you can pass `arima.sim` your own time series of random errors or the name of a function that will generate the errors (*e.g.*, you could use `rpois` if you wanted a model with Poisson errors). Type `?arima.sim` for more details.

Let's begin by simulating some AR(1) models and comparing their behavior. First, let's choose models with contrasting AR coefficients. Recall that in order for an AR(1) model to be stationary, $\phi < |1|$, so we'll try 0.1 and 0.9. We'll again set the random number seed so we will get the same answers.

```
set.seed(456)
## list description for AR(1) model with small coef
AR.sm <- list(order=c(1,0,0), ar=0.1, sd=0.1)
## list description for AR(1) model with large coef
AR.lg <- list(order=c(1,0,0), ar=0.9, sd=0.1)
## simulate AR(1)
AR1.sm <- arima.sim(n=50, model=AR.sm)
AR1.lg <- arima.sim(n=50, model=AR.lg)
```

Now let's plot the 2 simulated series.

```
## setup plot region
par(mfrow=c(1,2))
## get y-limits for common plots
ylm <- c(min(AR1.sm,AR1.lg), max(AR1.sm,AR1.lg))
## plot the ts
plot.ts(AR1.sm, ylim=ylm,
        ylab=expression(italic(x)[italic(t)]),
        main=expression(paste(phi," = 0.1")))
plot.ts(AR1.lg, ylim=ylm,
        ylab=expression(italic(x)[italic(t)]),
        main=expression(paste(phi," = 0.9")))
```

What do you notice about the two plots in Figure 1.22? It looks like the time series with the smaller AR coefficient is more "choppy" and seems to stay closer to 0 whereas the time series with the larger AR coefficient appears to wander around more. Remember that as the coefficient in an AR(1) model goes to 0, the model approaches a WN sequence, which is stationary in both the mean and variance. As the coefficient goes to 1, however, the model approaches a random walk, which is not stationary in either the mean or variance.

Next, let's generate two AR(1) models that have the same magnitude coeficient, but opposite signs, and compare their behavior.

```
set.seed(123)
## list description for AR(1) model with small coef
AR.pos <- list(order=c(1,0,0), ar=0.5, sd=0.1)
## list description for AR(1) model with large coef
AR.neg <- list(order=c(1,0,0), ar=-0.5, sd=0.1)
```

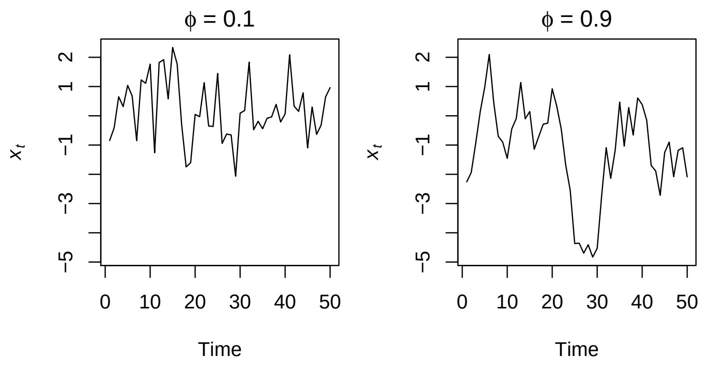

Fig. 1.22. Time series of simulated AR(1) processes with $\phi = 0.1$ (left) and $\phi = 0.9$ (right).

```
## simulate AR(1)
AR1.pos <- arima.sim(n=50, model=AR.pos)
AR1.neg <- arima.sim(n=50, model=AR.neg)
```

OK, let's plot the 2 simulated series.

```
## setup plot region
par(mfrow=c(1,2))
## get y-limits for common plots
ylm <- c(min(AR1.pos,AR1.neg), max(AR1.pos,AR1.neg))
## plot the ts
plot.ts(AR1.pos, ylim=ylm,
        ylab=expression(italic(x)[italic(t)]),
        main=expression(paste(phi[1]," = 0.5")))
plot.ts(AR1.neg,
        ylab=expression(italic(x)[italic(t)]),
        main=expression(paste(phi[1]," = -0.5")))
```

Now it appears like both time series vary around the mean by about the same amount, but the model with the negative coefficient produces a much more "sawtooth" time series. It turns out that any AR(1) model with $-1 < \phi < 0$ will exhibit the 2-point oscillation you see here.

We can simulate higher order AR(p) models in the same manner, but care must be exercised when choosing a set of coefficients that result in a stationary model or else `arima.sim` will fail and report an error. For example, an AR(2) model with both coefficients equal to 0.5 is not stationary, and therefore this function call will not work:

```
arima.sim(n=100, model=list(order(2,0,0), ar=c(0.5,0.5)))
```

If you try, R will respond that the "`'ar' part of model is not stationary`".

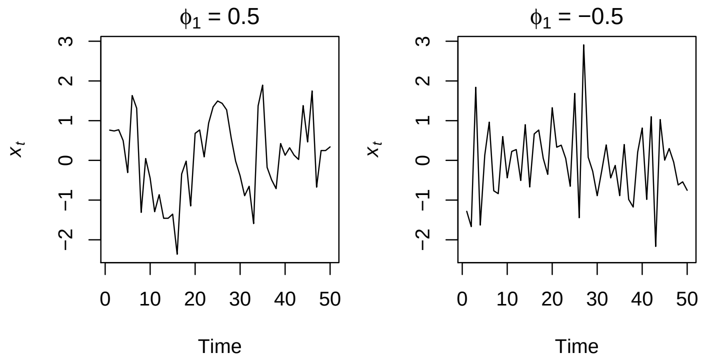

Fig. 1.23. Time series of simulated AR(1) processes with $\phi_1 = 0.5$ (left) and $\phi_1 = -0.5$ (right).

1.7.2 Correlation structure of AR(p) processes

Let's review what we learned in lecture about the general behavior of the ACF and PACF for AR(p) models. To do so, we'll simulate four stationary AR(p) models of increasing order p and then examine their ACF's and PACF's. Let's use a really big n so as to make them "pure", which will provide a much better estimate of the correlation structure.

```
set.seed(123)
## the 4 AR coefficients
ARp <- c(0.7, 0.2, -0.1, -0.3)
## empty list for storing models
AR.mods <- list()
## loop over orders of p
for(p in 1:4) {
  ## assume SD=1, so not specified
  AR.mods[[p]] <- arima.sim(n=10000, list(ar=ARp[1:p]))
}
```

Now that we have our four AR(p) models, lets look at plots of the time series, ACF's, and PACF's.

```
## set up plot region
par(mfrow=c(4,3))
## loop over orders of p
for(p in 1:4) {
  plot.ts(AR.mods[[p]][1:50],
          ylab=paste("AR(",p,")",sep=""))
  acf(AR.mods[[p]], lag.max=12)
  pacf(AR.mods[[p]], lag.max=12, ylab="PACF")
}
```

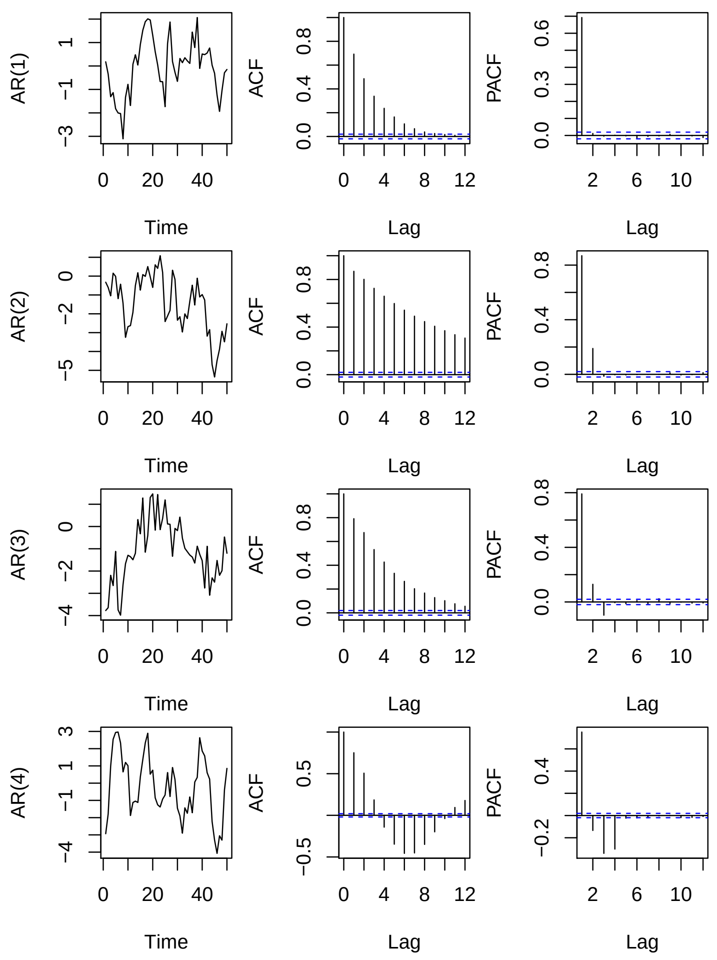

Fig. 1.24. Time series of simulated AR(p) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

As we saw in lecture and is evident from our examples shown in Figure 1.24, the ACF for an AR(p) process tails off toward zero very slowly, but the PACF goes to zero for lags $> p$. This is an important diagnostic tool when trying to identify the order of p in ARMA(p,q) models.

1.8 Moving-average (MA) models

A moving-averge process of order q, or MA(q), is a weighted sum of the current random error plus the q most recent errors, and can be written as

$$x_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q w_{t-q}, \tag{1.24}$$

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2; for our purposes we usually assume that $w_t \sim \text{N}(0,q)$. Of particular note is that because MA processes are finite sums of stationary errors, they themselves are stationary.

Of interest to us are so-called "invertible" MA processes that can be expressed as an infinite AR process with no error term. The term invertible comes from the inversion of the backshift operator ($\mathbf{B}$) that we discussed in class (*i.e.*, $\mathbf{B}x_t = x_{t-1}$). So, for example, an MA(1) process with $\theta < |1|$ is invertible because it can be written using the backshift operator as

$$\begin{aligned}
x_t &= w_t - \theta w_{t-1} \\
x_t &= w_t - \theta \mathbf{B} w_t \\
x_t &= (1 - \theta \mathbf{B}) w_t, \\
&\Downarrow \\
w_t &= \frac{1}{(1 - \theta \mathbf{B})} x_t \\
w_t &= (1 + \theta \mathbf{B} + \theta^2 \mathbf{B}^2 + \theta^3 \mathbf{B}^3 + \dots) x_t \\
w_t &= x_t + \theta x_{t-1} + \theta^2 x_{t-2} + \theta^3 x_{t-3} + \dots
\end{aligned} \tag{1.25}$$

1.8.1 Simulating an MA(q) process

We can simulate MA(q) processes just as we did for AR(p) processes using `arima.sim`. Here are 3 different ones with contrasting θ's:

```
set.seed(123)
## list description for MA(1) model with small coef
MA.sm <- list(order=c(0,0,1), ma=0.2, sd=0.1)
## list description for MA(1) model with large coef
MA.lg <- list(order=c(0,0,1), ma=0.8, sd=0.1)
## list description for MA(1) model with large coef
MA.neg <- list(order=c(0,0,1), ma=-0.5, sd=0.1)
## simulate MA(1)
MA1.sm <- arima.sim(n=50, model=MA.sm)
MA1.lg <- arima.sim(n=50, model=MA.lg)
MA1.neg <- arima.sim(n=50, model=MA.neg)
```

with their associated plots.

```
## setup plot region
par(mfrow=c(1,3))
## plot the ts
plot.ts(MA1.sm,
        ylab=expression(italic(x)[italic(t)]),
        main=expression(paste(theta," = 0.2")))
plot.ts(MA1.lg,
        ylab=expression(italic(x)[italic(t)]),
        main=expression(paste(theta," = 0.8")))
plot.ts(MA1.neg,
        ylab=expression(italic(x)[italic(t)]),
        main=expression(paste(theta," = -0.5")))
```

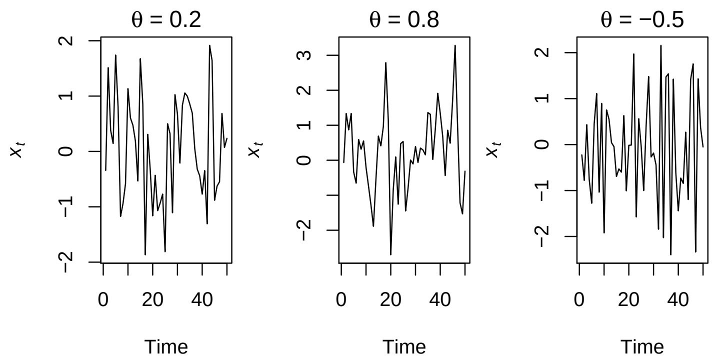

Fig. 1.25. Time series of simulated MA(1) processes with $\theta = 0.2$ (left), $\theta = 0.8$ (middle), and $\theta = -0.5$ (right).

In contrast to AR(1) processes, MA(1) models do not exhibit radically different behavior with changing θ. This should not be too surprising given that they are simply linear combinations of white noise.

1.8.2 Correlation structure of MA(q) processes

We saw in lecture and above how the ACF and PACF have distinctive features for AR(p) models, and they do for MA(q) models as well. Here are examples of four MA(q) processes. As before, we'll use a really big n so as to make them "pure", which will provide a much better estimate of the correlation structure.

```
set.seed(123)
## the 4 MA coefficients
MAq <- c(0.7, 0.2, -0.1, -0.3)
## empty list for storing models
```

```
MA.mods <- list()
## loop over orders of q
for(q in 1:4) {
  ## assume SD=1, so not specified
  MA.mods[[q]] <- arima.sim(n=1000, list(ma=MAq[1:q]))
}
```

Now that we have our four MA(q) models, lets look at plots of the time series, ACF's, and PACF's.

```
## set up plot region
par(mfrow=c(4,3))
## loop over orders of q
for(q in 1:4) {
  plot.ts(MA.mods[[q]][1:50],
          ylab=paste("MA(",q,")",sep=""))
  acf(MA.mods[[q]], lag.max=12)
  pacf(MA.mods[[q]], lag.max=12, ylab="PACF")
}
```

Note very little qualitative difference in the realizations of the four MA(q) processes (Figure 1.26). As we saw in lecture and is evident from our examples here, however, the ACF for an MA(q) process goes to zero for lags $> q$, but the PACF tails off toward zero very slowly. This is an important diagnostic tool when trying to identify the order of q in ARMA(p,q) models.

1.9 Autoregressive moving-average (ARMA) models

ARMA(p,q) models have a rich history in the time series literature, but they are not nearly as common in ecology as plain AR(p) models. As we discussed in lecture, both the ACF and PACF are important tools when trying to identify the appropriate order of p and q. Here we will see how to simulate time series from AR(p), MA(q), and ARMA(p,q) processes, as well as fit time series models to data based on insights gathered from the ACF and PACF.

We can write an ARMA(p,q) as a mixture of AR(p) and MA(q) models, such that

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t + \theta w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q x_{t-q}, \quad (1.26)$$

and the w_t are white noise.

1.9.1 Fitting ARMA(p,q) models with `arima`

We have already seen how to simulate AR(p) and MA(q) models with `arima.sim`; the same concepts apply to ARMA(p,q) models and therefore

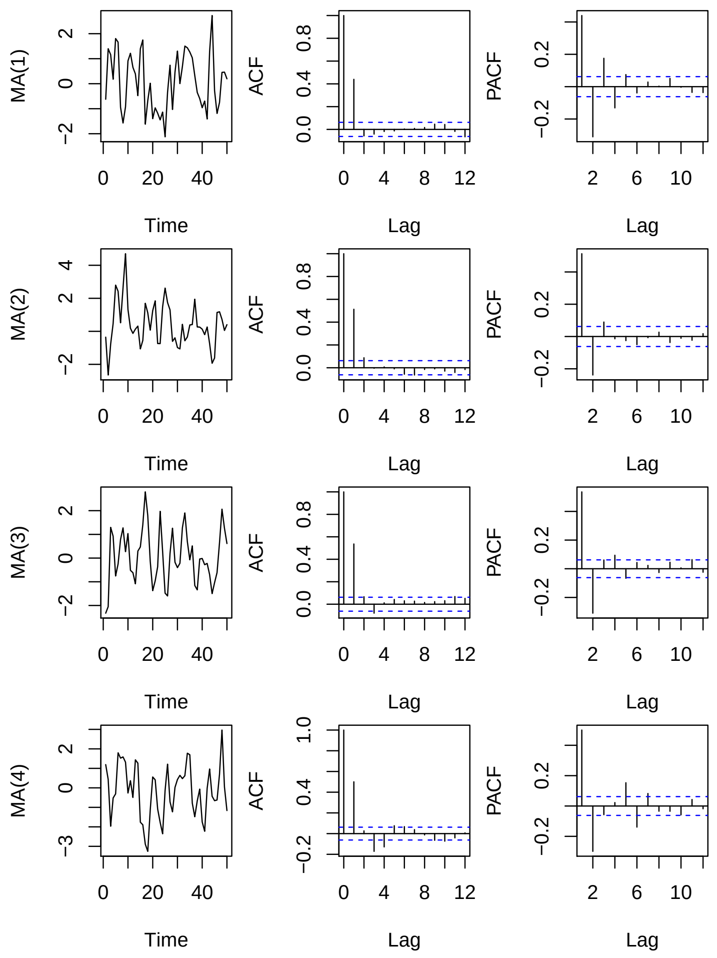

Fig. 1.26. Time series of simulated MA(q) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

we will not do that here. Instead, we will move on to fitting ARMA(p,q) models when we only have a realization of the process (*i.e.*, data) and do not know the underlying parameters that generated it.

The function `arima` accepts a number of arguments, but two of them are most important:

`x`	a univariate time series
`order`	a vector of length 3 specifying the order of ARIMA(p,d,q) model

In addition, note that by default `arima` will estimate an underlying mean of the time series unless $d > 0$. For example, an AR(1) process with mean μ would be written

$$x_t = \mu + \phi(x_{t-1} - \mu) + w_t. \tag{1.27}$$

If you know for a fact that the time series data have a mean of zero (*e.g.*, you already subtracted the mean from them), you should include the argument `include.mean=FALSE`, which is set to `TRUE` by default. Note that ignoring and not estimating a mean in ARMA(p,q) models when one exists will bias the estimates of all other parameters.

Let's see an example of how `arima` works. First we'll simulate an ARMA(2,2) model and then estimate the parameters to see how well we can recover them. In addition, we'll add in a constant to create a non-zero mean, which `arima` reports as `intercept` in its output.

```
set.seed(123)
## ARMA(2,2) description for arim.sim()
ARMA22 <- list(order=c(2,0,2), ar=c(-0.7,0.2), ma=c(0.7,0.2))
## mean of process
mu <- 5
## simulated process (+ mean)
ARMA.sim <- arima.sim(n=10000, model=ARMA22) + mu
## estimate parameters
arima(x=ARMA.sim, order=c(2,0,2))
```

```
Call:
arima(x = ARMA.sim, order = c(2, 0, 2))

Coefficients:
          ar1     ar2     ma1     ma2  intercept
      -0.7079  0.1924  0.6912  0.2001     4.9975
s.e.   0.0291  0.0284  0.0289  0.0236     0.0125

sigma^2 estimated as 0.9972:  log likelihood = -14175.92,  aic = 28363.84
```

It looks like we were pretty good at estimating the true parameters, but our sample size was admittedly quite large (the estimate of the variance of the process errors is reported as `sigma^2` below the other coefficients). As an exercise, try decreasing the length of time series in the `arima.sim` call above from 10,000 to something like 100 and see what effect it has on the parameter estimates.

1.9.2 Searching over model orders

In an ideal situation, you could examine the ACF and PACF of the time series of interest and immediately decipher what orders of p and q must have generated the data, but that doesn't always work in practice. Instead, we are often left with the task of searching over several possible model forms and seeing which of them provides the most parsimonious fit to the data. There are two easy ways to do this for ARIMA models in `R`. The first is to write a little script that loops ove the possible dimensions of p and q. Let's try that for the process we simulated above and search over orders of p and q from 0-3 (it will take a few moments to run and will likely report an error about a "`possible convergence problem`", which you can ignore).

```
## empty list to store model fits
ARMA.res <- list()
## set counter
cc <- 1
## loop over AR
for(p in 0:3) {
  ## loop over MA
  for(q in 0:3) {
    ARMA.res[[cc]] <- arima(x=ARMA.sim,order=c(p,0,q))
    cc <- cc + 1
  }
}
## get AIC values for model evaluation
ARMA.AIC <- sapply(ARMA.res,function(x) x$aic)
## model with lowest AIC is the best
ARMA.res[[which(ARMA.AIC==min(ARMA.AIC))]]
```

```
Call:
arima(x = ARMA.sim, order = c(p, 0, q))

Coefficients:
          ar1     ar2     ma1     ma2  intercept
      -0.7079  0.1924  0.6912  0.2001     4.9975
s.e.   0.0291  0.0284  0.0289  0.0236     0.0125

sigma^2 estimated as 0.9972:  log likelihood = -14175.92,  aic = 28363.84
```

It looks like our search worked, so let's look at the other method for fitting ARIMA models. The `auto.arima` function in the package `forecast` will conduct an automatic search over all possible orders of ARIMA models that you specify. For details, type `?auto.arima` after loading the package. Let's repeat our search using the same criteria.

```
## (install if necessary) & load forecast pkg
if(!require("forecast")) {
    install.packages("forecast")
    library("forecast")
```

```
 }
 ## find best ARMA(p,q) model
 auto.arima(ARMA.sim, start.p=0, max.p=3, start.q=0, max.q=3)
```

```
Series: ARMA.sim
ARIMA(2,0,2) with non-zero mean

Coefficients:
          ar1     ar2     ma1     ma2  intercept
      -0.7079  0.1924  0.6912  0.2001     4.9975
s.e.   0.0291  0.0284  0.0289  0.0236     0.0125

sigma^2 estimated as 0.9977:  log likelihood=-14175.92
AIC=28363.84   AICc=28363.84   BIC=28407.1
```

We get the same results with an increase in speed and less coding, which is nice. If you want to see the form for each of the models checked by `auto.arima` and their associated AIC values, include the argument `trace=1`.

Problems

We have seen how to do a variety of introductory time series analyses with R. Now it is your turn to apply the information you learned here and in lecture to complete some analyses. You have been asked by a colleauge to help analyze some time series data she collected as part of an experiment on the effects of light and nutrients on the population dynamics of phytoplankton. Specifically, after controlling for differences in light and temperature, she wants to know if the natural log of population density can be modeled with some form of ARMA(p,q) model. The data are expressed as the number of cells per milliliter recorded every hour for one week beginning at 8:00 AM on December 1, 2014; you can find them here:

```
## get phytoplankton data
pp <- "http://faculty.washington.edu/scheuerl/phytoDat.txt"
pDat <- read.table(pp)
```

Use the information above to do the following:

1.1 Convert `pDat`, which is a `data.frame` object, into a `ts` object. This bit of code might be useful to get you started:

```
## what day of 2014 is Dec 1st?
dBegin <- as.Date("2014-12-01")
dayOfYear <- (dBegin - as.Date("2014-01-01") + 1)
```

1.2 Plot the time series of phytoplankton density and provide a brief description of any notable features.

1.3 Although you do not have the actual measurements for the specific temperature and light regimes used in the experiment, you have been informed that they follow a regular light/dark period with accompanying warm/cool temperatures. Thus, estimating a fixed seasonal effect is justifiable. Also, the instrumentation is precise enough to preclude any systematic change in measurements over time (*i.e.*, you can assume $m_t = 0$ for all t). Obtain the time series of the estimated log-density of phytoplankton absent any hourly effects caused by variation in temperature or light. (*Hint*: You will need to do some decomposition.)

1.4 Use diagnostic tools to identify the possible order(s) of ARMA model(s) that most likely describes the log of population density for this particular experiment. Note that at this point you should be focusing your analysis on the results obtained in Question 3.

1.5 Use some form of search to identify what form of ARMA(p,q) model best describes the log of population density for this particular experiment. Use what you learned in Question 4 to inform possible orders of p and q. (*Hint*: if you use `auto.arima`, include the additional argument `seasonal=FALSE`)

1.6 Write out the best model in the form of Equation (1.26) using the underscore notation to refer to subscripts (*e.g.*, write `x_t` for x_t). You can round any parameters/coefficients to the nearest hundreth. (***Hint***: if the mean of the time series is not zero, refer to Eqn 1.27 in the lab handout).