Applied Time Series Analysis for Fisheries and Environmental Sciences

E. E. Holmes, M. D. Scheuerell, and E. J. Ward

2019-11-26

Contents

1 Basic matrix math in R **11**

1.1 Creating matrices in R 11
1.2 Matrix multiplication, addition and transpose 14
1.3 Subsetting a matrix 16
1.4 Replacing elements in a matrix 18
1.5 Diagonal matrices and identity matrices 20
1.6 Taking the inverse of a square matrix 22
1.7 Problems 24

2 Linear regression in matrix form **27**

2.1 A simple regression: one explanatory variable 28
2.2 Matrix Form 1 29
2.3 Matrix Form 2 33
2.4 Groups of intercepts 38
2.5 Groups of β's 41
2.6 Seasonal effect as a factor 44
2.7 Seasonal effect plus other explanatory variables* 47
2.8 Models with confounded parameters* 48
2.9 Problems 51

3 Introduction to time series **53**

3.1 Examples of time series 53
3.2 Classification of time series 55
3.3 Statistical analyses of time series 56
3.4 What is a time series model? 57
3.5 Two simple and classic time series models 57
3.6 Classical decomposition 59

3.7 Decomposition on log-transformed data 65

4 Basic time series functions in R **69**
4.1 Time series plots 70
4.2 Decomposition of time series 73
4.3 Differencing to remove a trend or seasonal effects 81
4.4 Correlation within and among time series 83
4.5 White noise (WN) 93
4.6 Random walks (RW) 97
4.7 Autoregressive (AR) models 100
4.8 Moving-average (MA) models 105
4.9 Autoregressive moving-average (ARMA) models 109
4.10 Problems 115

5 Box-Jenkins method **117**
5.1 Box-Jenkins method 118
5.2 Stationarity 118
5.3 Dickey-Fuller and Augmented Dickey-Fuller tests 126
5.4 KPSS test 132
5.5 Dealing with non-stationarity 134
5.6 Summary: stationarity testing 137
5.7 Estimating ARMA parameters 138
5.8 Estimating the ARMA orders 143
5.9 Check residuals 148
5.10 Forecast from a fitted ARIMA model 149
5.11 Seasonal ARIMA model 150
5.12 Forecast using a seasonal model 152
5.13 Problems 153

6 Univariate state-space models **157**
6.1 Fitting a state-space model with MARSS 158
6.2 Examples using the Nile river data 160
6.3 The StructTS function 164
6.4 Comparing models with AIC and model weights 167
6.5 Basic diagnostics 168
6.6 Fitting with JAGS 171
6.7 Fitting with Stan 174
6.8 A random walk model of animal movement 179

6.9 Problems 180

7 MARSS models **187**
7.1 Overview 188
7.2 West coast harbor seals counts 188
7.3 A single well-mixed population 191
7.4 Four subpopulations with temporally uncorrelated errors . . . 195
7.5 Four subpopulations with temporally correlated errors 197
7.6 Using MARSS models to study spatial structure 200
7.7 Hypotheses regarding spatial structure 200
7.8 Set up the hypotheses as different models 202
7.9 Fitting a MARSS model with JAGS 205
7.10 Fitting a MARSS model with Stan 208
7.11 Problems 211

8 MARSS models with covariates **217**
8.1 Overview 218
8.2 Prepare the plankton data 218
8.3 Observation-error only model 219
8.4 Process-error only model 221
8.5 Both process- and observation-error 224
8.6 Including seasonal effects in MARSS models 227
8.7 Model diagnostics 233
8.8 Homework data and discussion 234
8.9 Problems 237

9 Dynamic linear models **239**
9.1 Overview 240
9.2 DLM in state-space form 241
9.3 Stochastic level models 242
9.4 Stochastic regression model 244
9.5 DLM with seasonal effect 245
9.6 Analysis of salmon survival 247
9.7 Fitting with `MARSS()` 248
9.8 Forecasting 251
9.9 Forecast diagnostics 257
9.10 Homework discussion and data 260
9.11 Problems 263

10 Dynamic Factor Analysis **265**
10.1 Introduction . . . 266
10.2 Example of a DFA model . . . 266
10.3 Constraining a DFA model . . . 267
10.4 Different error structures . . . 269
10.5 Lake Washington phytoplankton data . . . 269
10.6 Fitting DFA models with the MARSS package . . . 272
10.7 Interpreting the MARSS output . . . 275
10.8 Rotating trends and loadings . . . 275
10.9 Estimated states and loadings . . . 276
10.10 Plotting the data and model fits . . . 279
10.11 Covariates in DFA models . . . 281
10.12 Example from Lake Washington . . . 281
10.13 Problems . . . 287

11 Covariates with Missing Values **289**
11.1 Covariates with missing values or observation error . . . 290
11.2 Example: Snotel Data . . . 292
11.3 Modeling Seasonal SWE . . . 304

12 JAGS for Bayesian time series analysis **309**
12.1 The airquality dataset . . . 310
12.2 Linear regression with no covariates . . . 310
12.3 Regression with autocorrelated errors . . . 317
12.4 Random walk time series model . . . 319
12.5 Autoregressive AR(1) time series models . . . 322
12.6 Univariate state space model . . . 323
12.7 Forecasting with JAGS models . . . 325
12.8 Problems . . . 327

13 Stan for Bayesian time series analysis **329**
13.1 Linear regression . . . 330
13.2 Linear regression with correlated errors . . . 333
13.3 Random walk model . . . 335
13.4 Autoregressive models . . . 335
13.5 Univariate state-space models . . . 336
13.6 Dynamic factor analysis . . . 337
13.7 Uncertainty intervals on states . . . 341

Chapter 3

Introduction to time series

At a very basic level, a time series is a set of observations taken sequentially in time. It is different than non-temporal data because each data point has an order and is, typically, related to the data points before and after by some process.

A script with all the R code in the chapter can be downloaded here. The Rmd for this chapter can be downloaded here.

3.1 Examples of time series

```
data(WWWusage, package="datasets")
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
plot.ts(WWWusage, ylab = "", las = 1, col = "blue", lwd = 2)
```

```
data(lynx, package="datasets")
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
plot.ts(lynx, ylab = "", las = 1, col = "blue", lwd = 2)
```

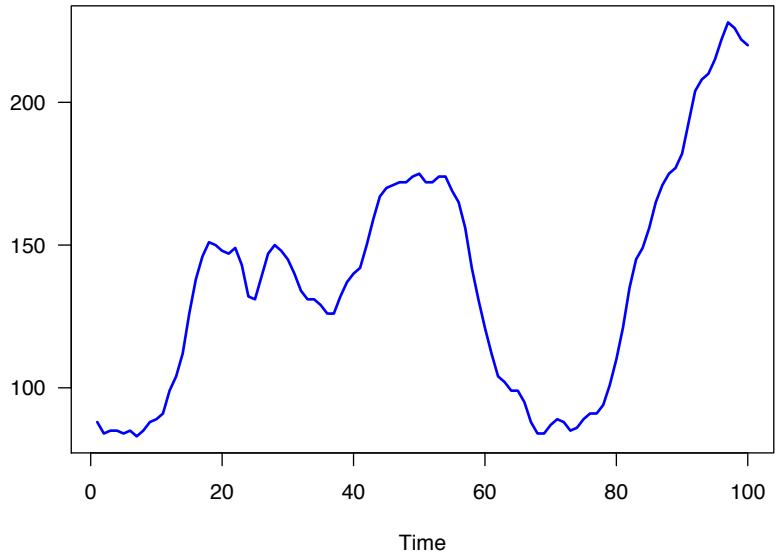


Figure 3.1: Number of users connected to the internet

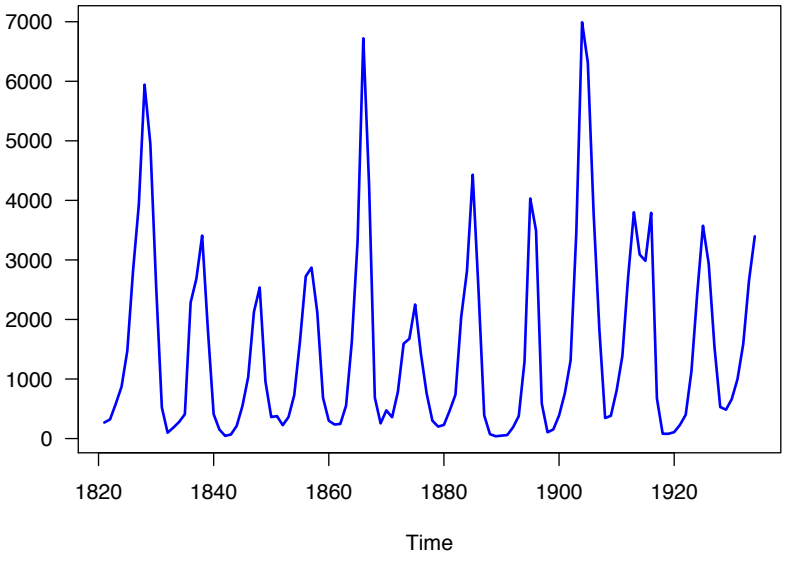


Figure 3.2: Number of lynx trapped in Canada from 1821-1934

3.2 Classification of time series

A ts can be represented as a set

$$\{x_1, x_2, x_3, \ldots, x_n\}$$

For example,

$$\{10, 31, 27, 42, 53, 15\}$$

It can be further classified.

3.2.1 By some *index set*

Interval across real time; $x(t)$

- begin/end: $t \in [1.1, 2.5]$

Discrete time; x_t

- Equally spaced: $t = \{1, 2, 3, 4, 5\}$
- Equally spaced w/ missing value: $t = \{1, 2, 4, 5, 6\}$
- Unequally spaced: $t = \{2, 3, 4, 6, 9\}$

3.2.2 By the *underlying process*

Discrete (eg, total # of fish caught per trawl)

Continuous (eg, salinity, temperature)

3.2.3 By the *number of values recorded*

Univariate/scalar (eg, total # of fish caught)

Multivariate/vector (eg, # of each spp of fish caught)

3.2.4 By the *type of values recorded*

Integer (eg, # of fish in 5 min trawl = 2413)

Rational (eg, fraction of unclipped fish = 47/951)

Real (eg, fish mass = 10.2 g)

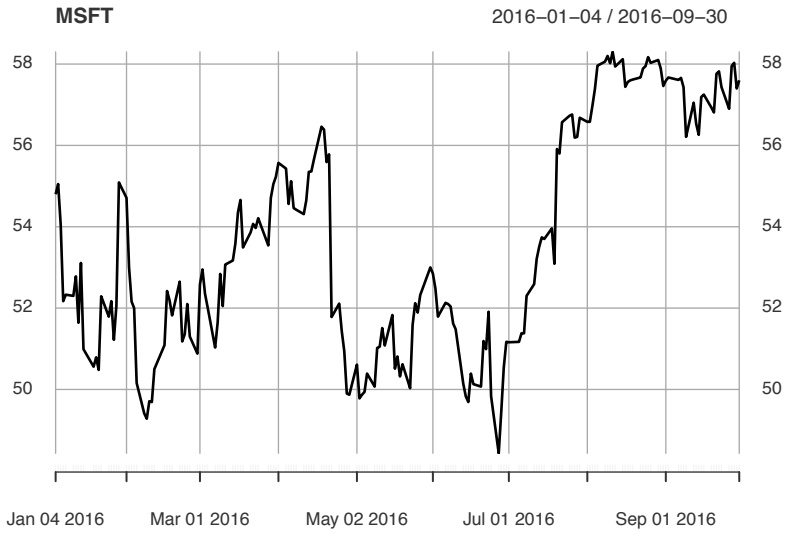
Complex (eg, cos(2 2.43) + i sin(2 2.43))

3.3 Statistical analyses of time series

Most statistical analyses are concerned with estimating properties of a population from a sample. For example, we use fish caught in a seine to infer the mean size of fish in a lake. Time series analysis, however, presents a different situation:

- Although we could vary the length of an observed time series, it is often impossible to make multiple observations at a given point in time

For example, one can't observe today's closing price of Microsoft stock more than once. Thus, conventional statistical procedures, based on large sample estimates, are inappropriate.



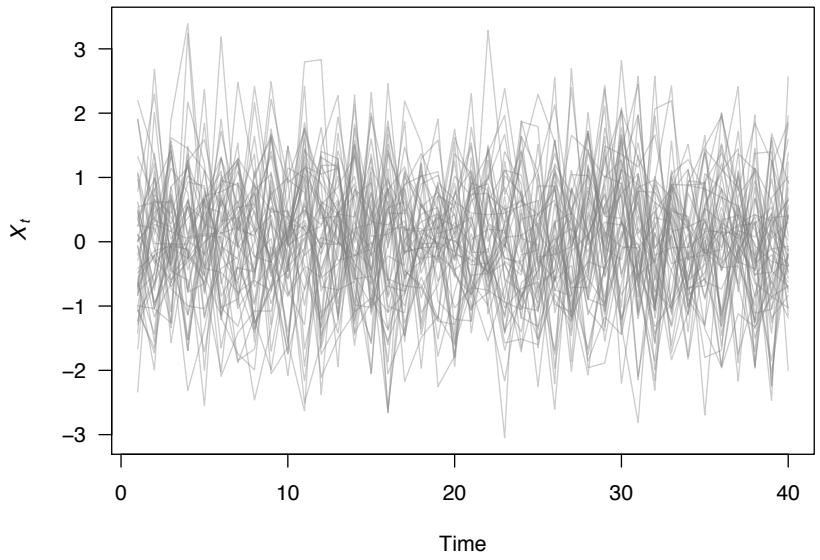


Figure 3.3: Distribution of realizations

3.4 What is a time series model?

We use a time series model to analyze time series data. A time series model for $\{x_t\}$ is a specification of the joint distributions of a sequence of random variables $\{X_t\}$, of which $\{x_t\}$ is thought to be a realization.

Here is a plot of many realizations from a time series model.

These lines represent the distribution of possible realizations. However, we have only one realization. The time series model allows us to use the one realization we have to make inferences about the underlying joint distribution from whence our realization came.

3.5 Two simple and classic time series models

White noise: $x_t \sim N(0, 1)$

```
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
matplot(ww, type="l", lty="solid",  las = 1,
```

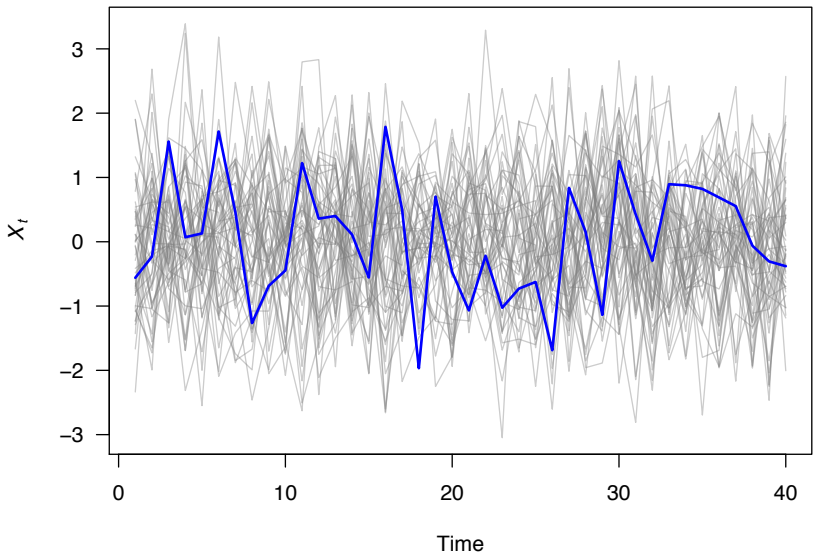
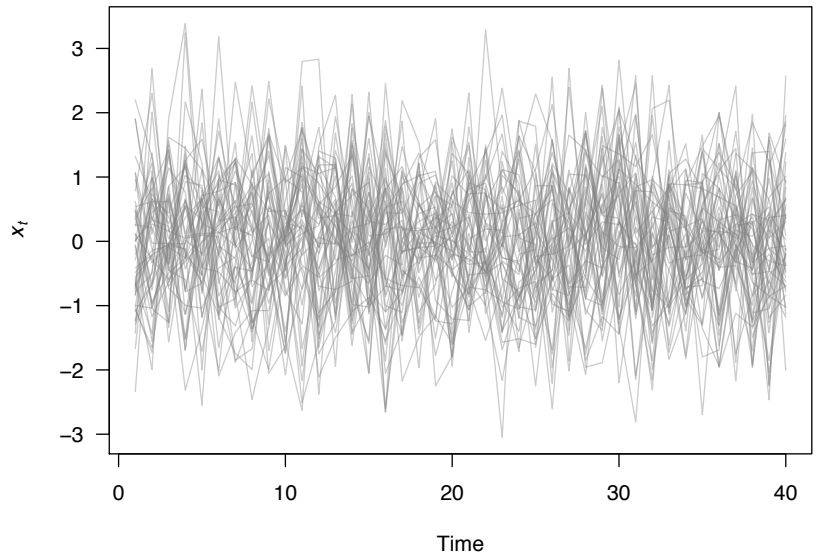


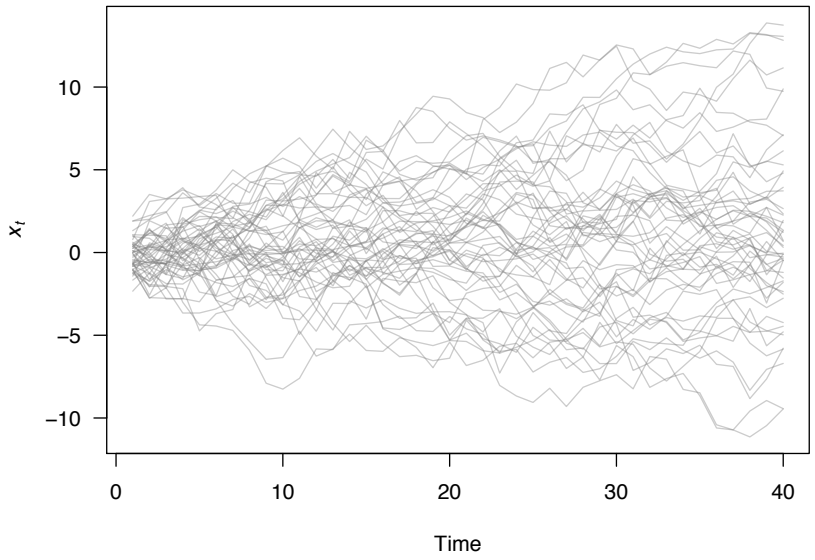
Figure 3.4: Blue line is our one realization.

```
        ylab = expression(italic(x[t])), xlab = "Time",
        col = gray(0.5, 0.4))
```



Random walk: $x_t = x_{t-1} + w_t,$ with $w_t \sim N(0, 1)$

```
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
matplot(apply(ww, 2, cumsum), type="l", lty="solid",  las = 1,
        ylab = expression(italic(x[t])), xlab = "Time",
        col = gray(0.5, 0.4))
```



3.6 Classical decomposition

Model time series $\{x_t\}$ as a combination of

1. trend (m_t)
2. seasonal component (s_t)
3. remainder (e_t)

$x_t = m_t + s_t + e_t$

3.6.1 1. The trend (m_t)

We need a way to extract the so-called signal. One common method is via "linear filters"

$$m_t = \sum_{i=-\infty}^{\infty} \lambda_i x_{t+1}$$

For example, a moving average

$$m_t = \sum_{i=-a}^{a} \frac{1}{2a+1} x_{t+i}$$

If $a = 1$, then

$$m_t = \frac{1}{3}(x_{t-1} + x_t + x_{t+1})$$

3.6.2 Example of linear filtering

Here is a time series.

A linear filter with $a = 3$ closely tracks the data.

As we increase the length of data that is averaged from 1 on each side ($a = 3$) to 4 on each side ($a = 9$), the trend line is smoother.

When we increase up to 13 points on each side ($a = 27$), the trend line is very smooth.

3.6.3 2. Seasonal effect (s_t)

Once we have an estimate of the trend m_t, we can estimate s_t simply by subtraction:

$$s_t = x_t - m_t$$

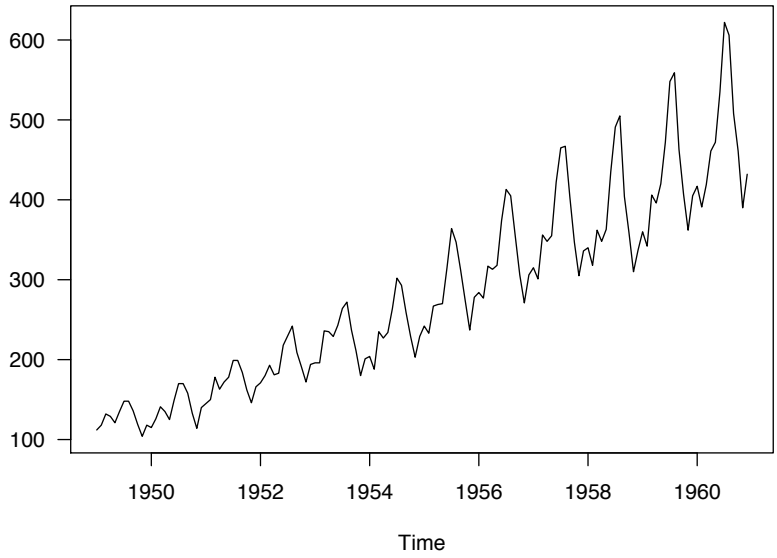


Figure 3.5: Monthly airline passengers from 1949-1960

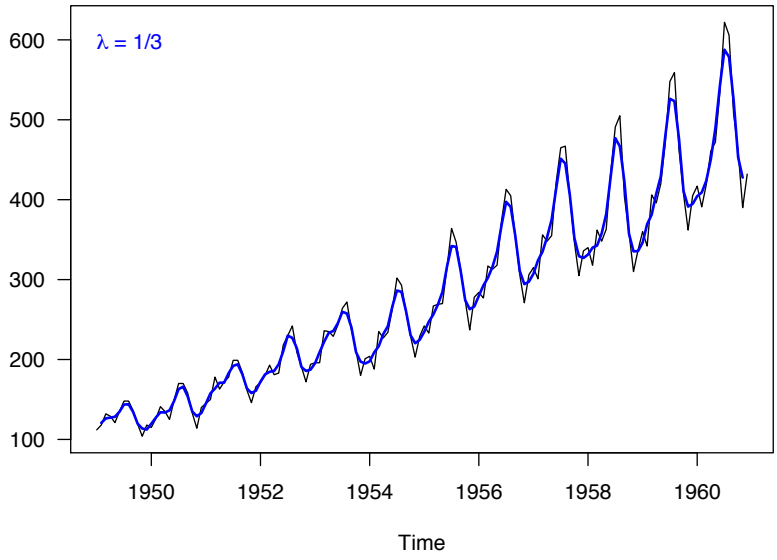


Figure 3.6: Monthly airline passengers from 1949-1960 with a low filter.

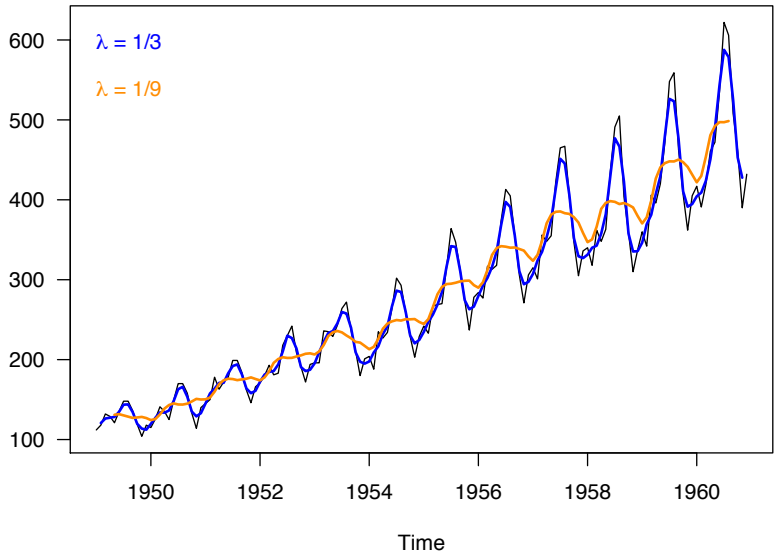


Figure 3.7: Monthly airline passengers from 1949-1960 with a medium filter.

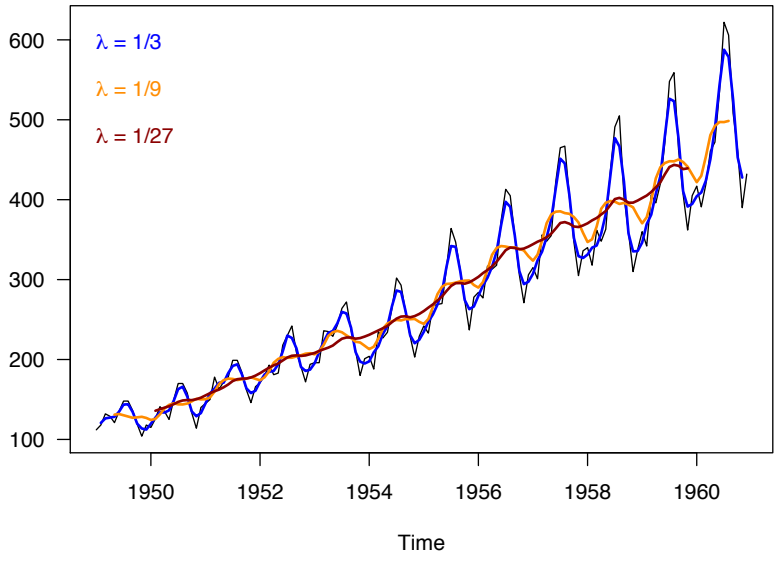
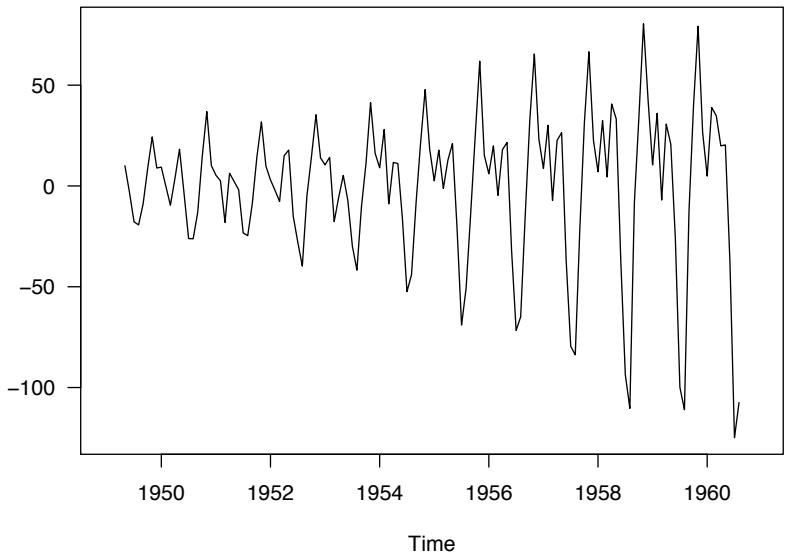


Figure 3.8: Monthly airline passengers from 1949-1960 with a high filter.



This is the seasonal effect (s_t), assuming $\lambda = 1/9$, but, s_t includes the remainder e_t as well. Instead we can estimate the mean seasonal effect (s_t).

```
seas_2 <- decompose(xx)$seasonal
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
plot.ts(seas_2, las = 1, ylab = "")
```

3.6.4 3. Remainder (e_t)

Now we can estimate e_t via subtraction:

$$e_t = x_t - m_t - s_t$$

```
ee <- decompose(xx)$random
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
plot.ts(ee, las = 1, ylab = "")
```

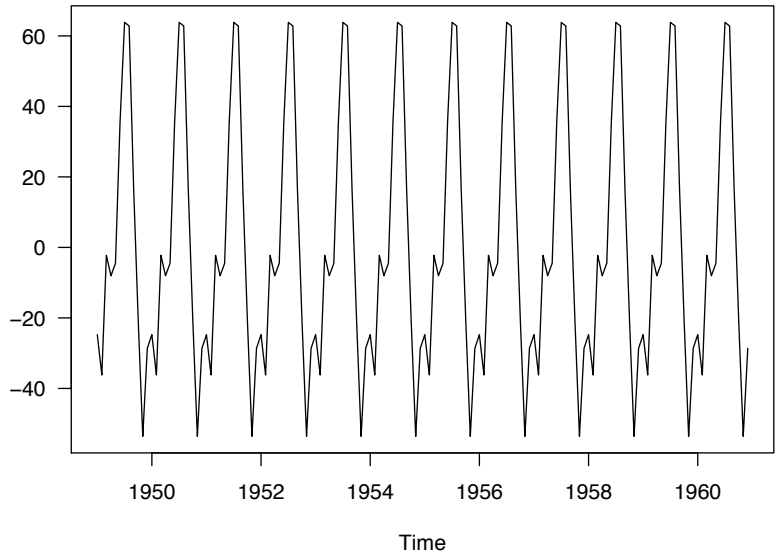


Figure 3.9: Mean seasonal effect.

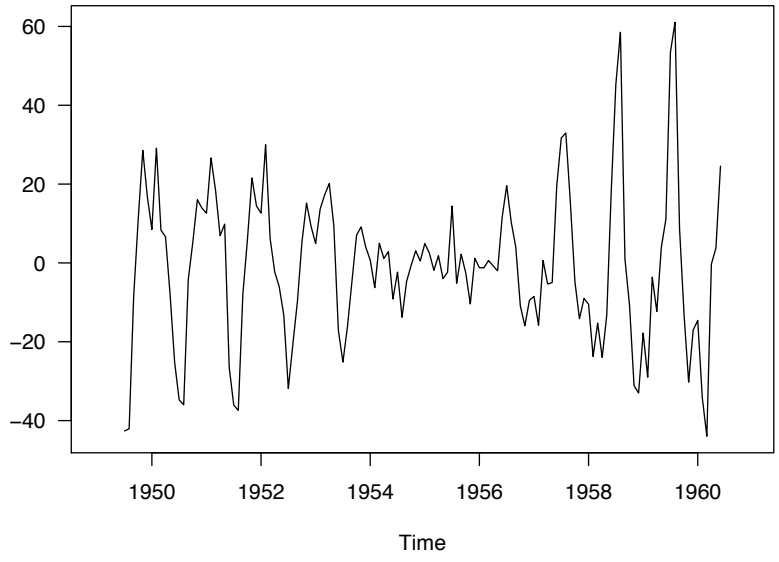


Figure 3.10: Errors.

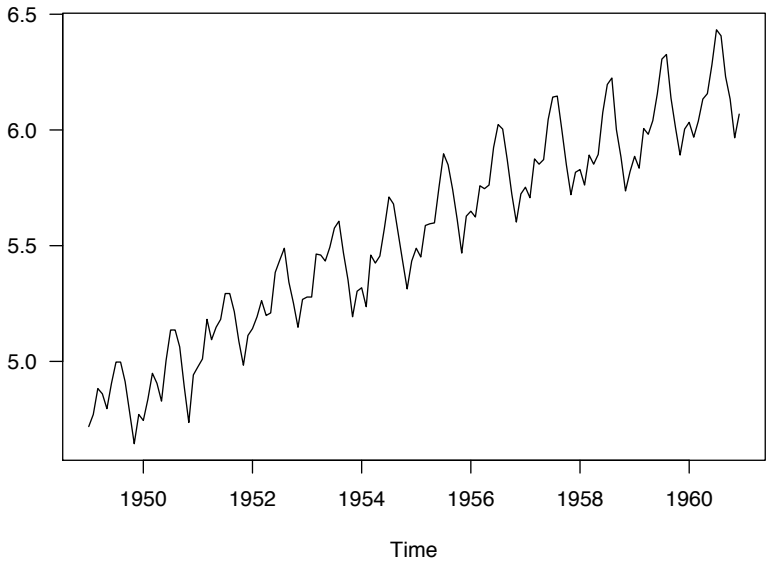


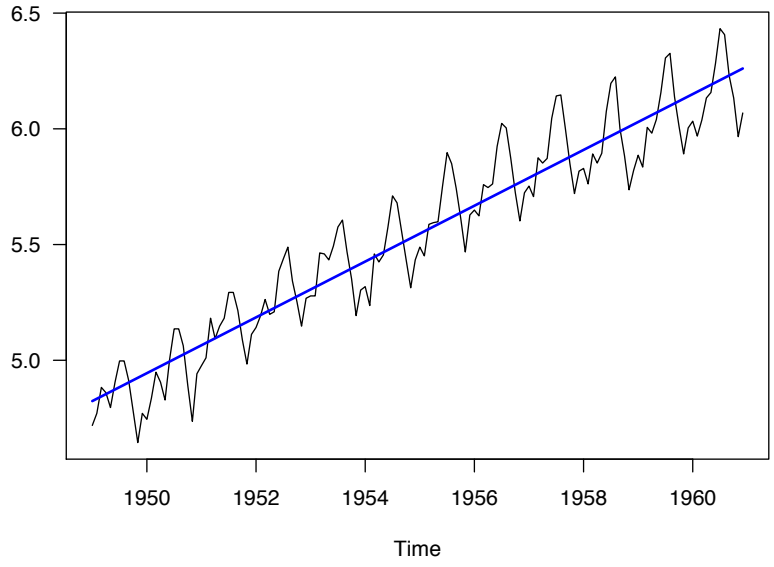
Figure 3.11: Log monthly airline passengers from 1949-1960

3.7 Decomposition on log-transformed data

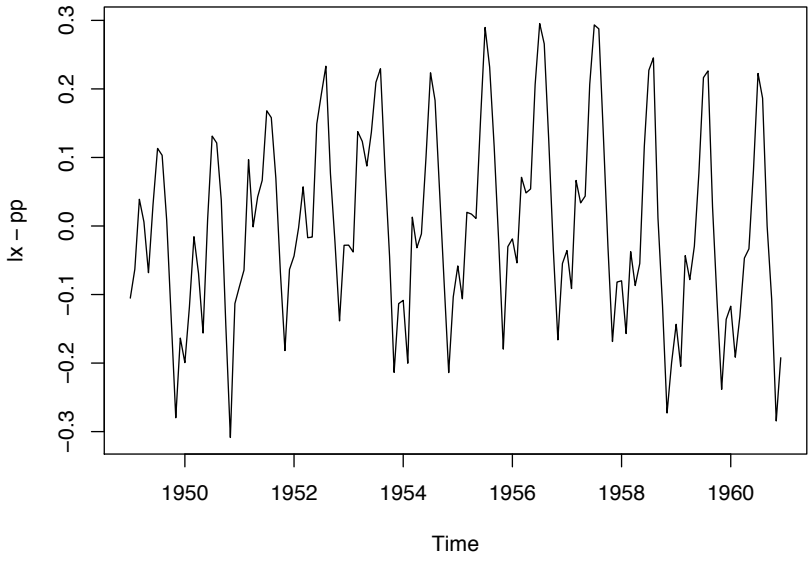
Let's repeat the decomposition with the log of the airline data.

```
lx <- log(AirPassengers)
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
plot.ts(lx, las = 1, ylab = "")
```

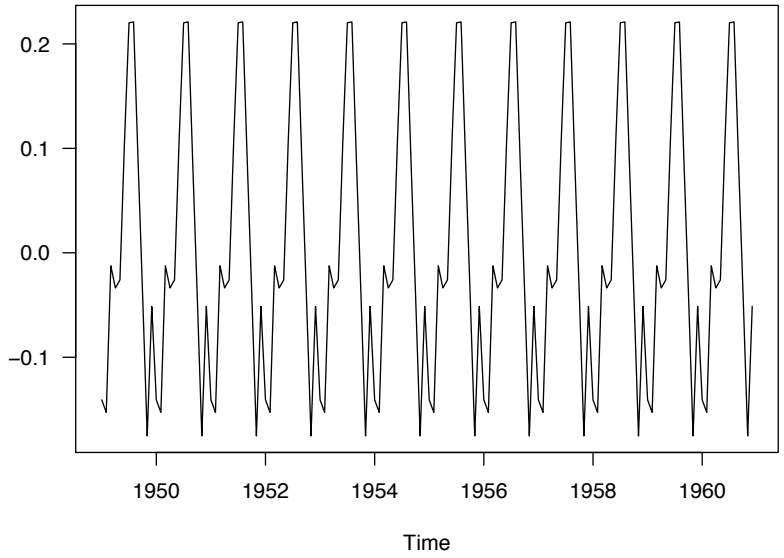
3.7.1 The trend (m_t)



3.7.2 Seasonal effect (s_t) with error (e_t)

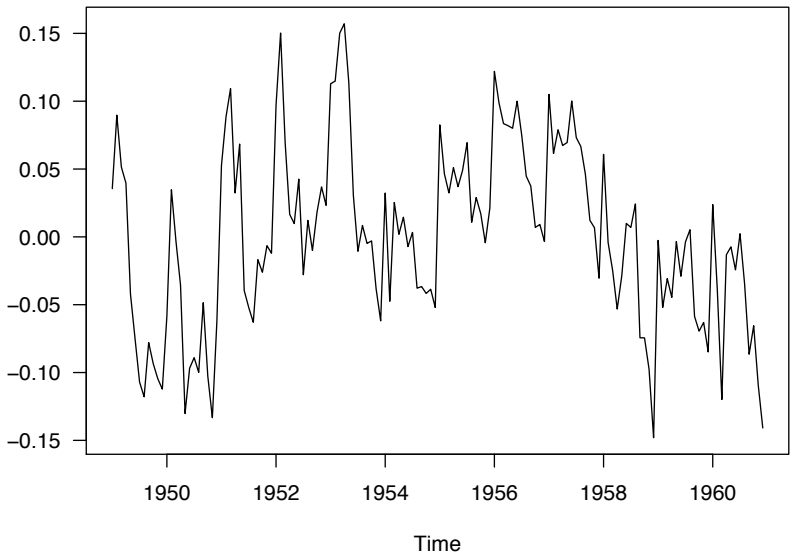


3.7.3 Mean seasonal effect (s_t)



3.7.4 Remainder (e_t)

```
le <- lx - pp - seas_2
par(mai = c(0.9,0.9,0.1,0.1), omi = c(0,0,0,0))
plot.ts(le, las = 1, ylab = "")
```


0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
1950
1952
1954
1956
1958
1960
Time

Chapter 4

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.

A script with all the R code in the chapter can be downloaded here. The Rmd for this chapter can be downloaded here.

Data and packages

This chapter uses the **stats** package, which is often loaded by default when you start R, the **MARSS** package and the **forecast** package. The problems use a dataset in the **datasets** package. After installing the packages, if needed, load:

```
library(stats)
library(MARSS)
library(forecast)
library(datasets)
```

The chapter uses data sets which are in the **atsalibrary** package. If needed, install using the **devtools** package.

```
library(devtools)
devtools::install_github("nwfsc-timeseries/atsalibrary")
```

The main one is a time series of the atmospheric concentration of CO_2 collected at the Mauna Loa Observatory in Hawai'i (`MLCO2`). The second is Northern Hemisphere land and ocean temperature anomalies from NOAA. (`NHTemp`). The problems use a data set on hourly phytoplankton counts (`hourlyphyto`). Use `?MLCO2`, `?NHTemp` and `?hourlyphyto` for information on these datasets.

Load the data.

```
data(NHTemp, package="atsalibrary")
Temp <- NHTemp
data(MLCO2, package="atsalibrary")
CO2 <- MLCO2
data(hourlyphyto, package="atsalibrary")
pDat <- hourlyphyto
```

4.1 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).

4.1.1 ts objects and `plot.ts()`

The CO_2 data are stored in R as a `data.frame` object, 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.

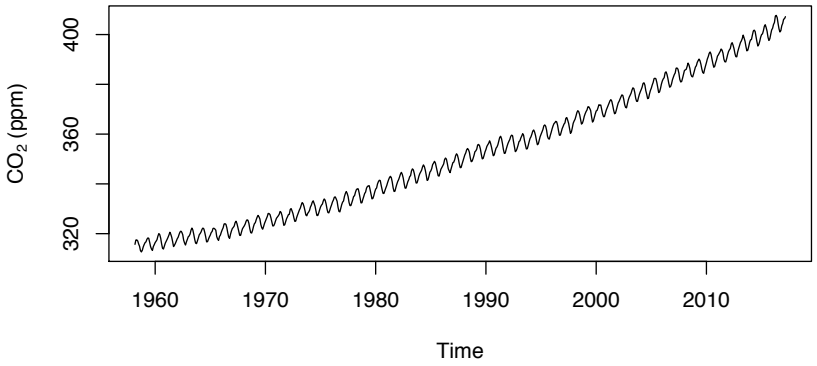


Figure 4.1: Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i measured monthly from March 1958 to present.

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)")))
```

Examination of the plotted time series (Figure 4.1) 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?)

4.1.2 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 use the data on monthly mean temperature anomolies for the Northern Hemisphere (`Temp`). First convert `Temp` to a `ts` object.

```
temp.ts <- 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,temp.ts)
## dimensions of common-time data
dim(datI)
```

```
[1] 682   2
```

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

```
[1] 1647    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 rec-

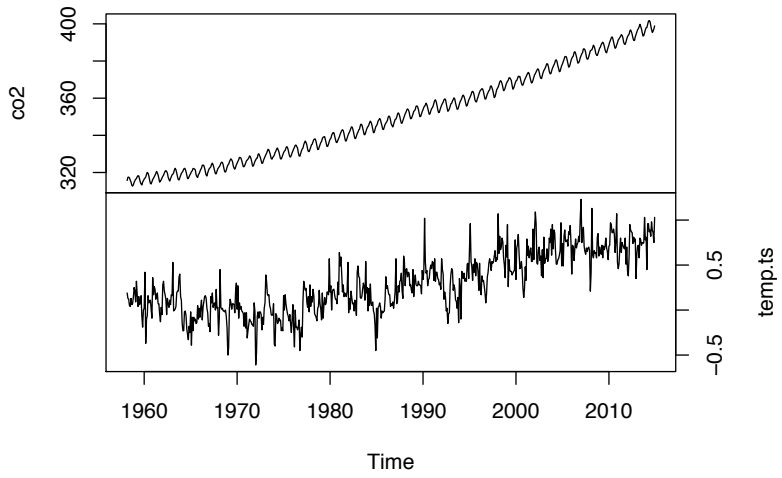


Figure 4.2: Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) and the mean temperature index for the Northern Hemisphere (bottom) measured monthly from March 1958 to present.

ognize 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 4.2):

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

4.2 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

$$x_t = m_t + s_t + e_t, \tag{4.1}$$

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 $\{e_t\}$.

4.2.1 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

$$\hat{m}_t = \sum_{k=-a}^{a} \left(\frac{1}{1+2a}\right) x_{t+k}. \tag{4.2}$$

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

$$\hat{m}_t = \frac{\frac{1}{2}x_{t-6} + x_{t-5} + \cdots + x_{t-1} + x_t + x_{t+1} + \cdots + x_{t+5} + \frac{1}{2}x_{t+6}}{12} \tag{4.3}$$

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

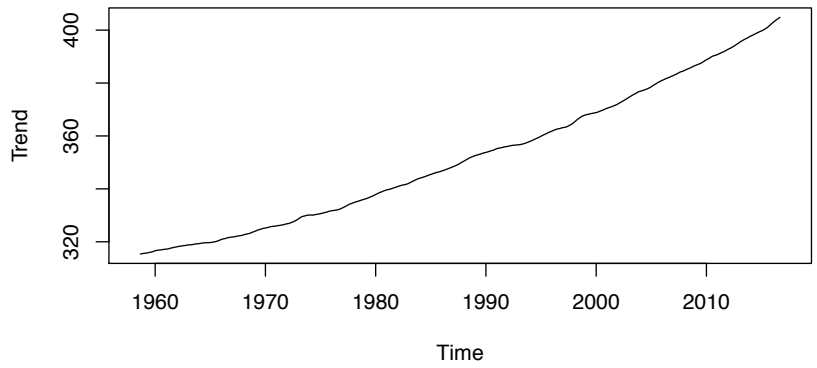


Figure 4.3: Time series of the estimated trend $\{\hat{m}_t\}$ for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

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 4.3).

4.2.2 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

$$\hat{s}_t = x_t - \hat{m}_t, \tag{4.4}$$

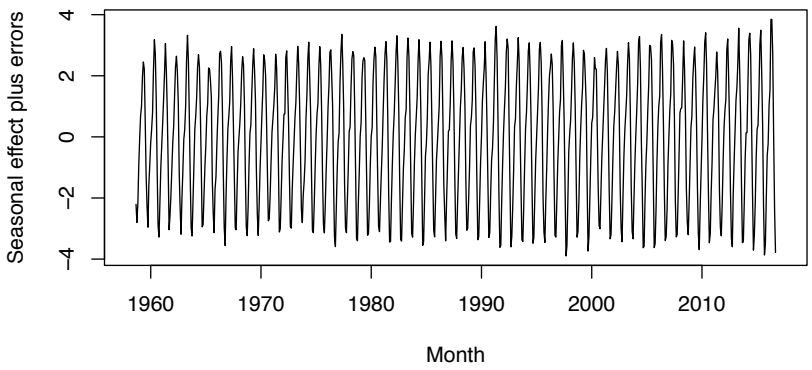


Figure 4.4: Time series of seasonal effects plus random errors for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i, measured monthly from March 1958 to present.

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 (4.1) and (4.4).

```
## 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)
```

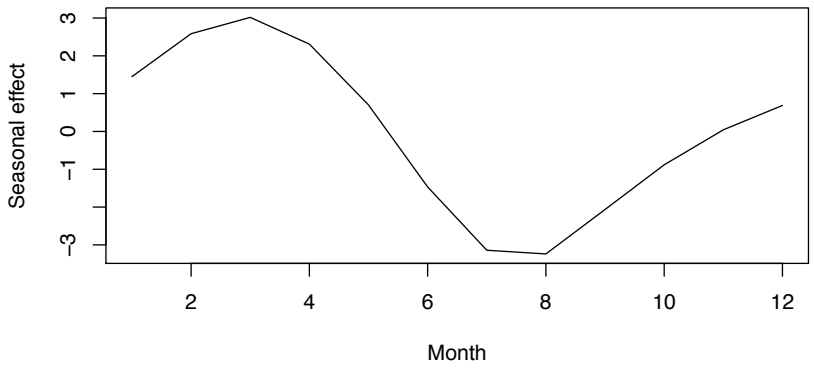


Figure 4.5: Estimated monthly seasonal effects for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

```
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 4.5). (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)
```

4.2.3 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

$$\hat{e}_t = x_t - \hat{m}_t - \hat{s}_t. \tag{4.5}$$

Again, this is really easy in R:

```
## 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 4.6.

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

4.2.4 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 $\{\hat{e}_t\}$

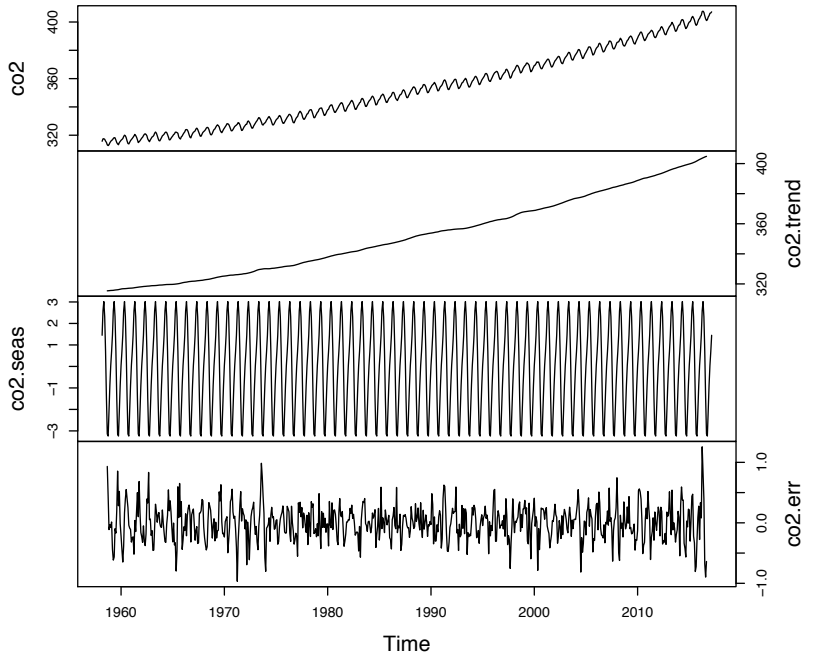


Figure 4.6: Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors.

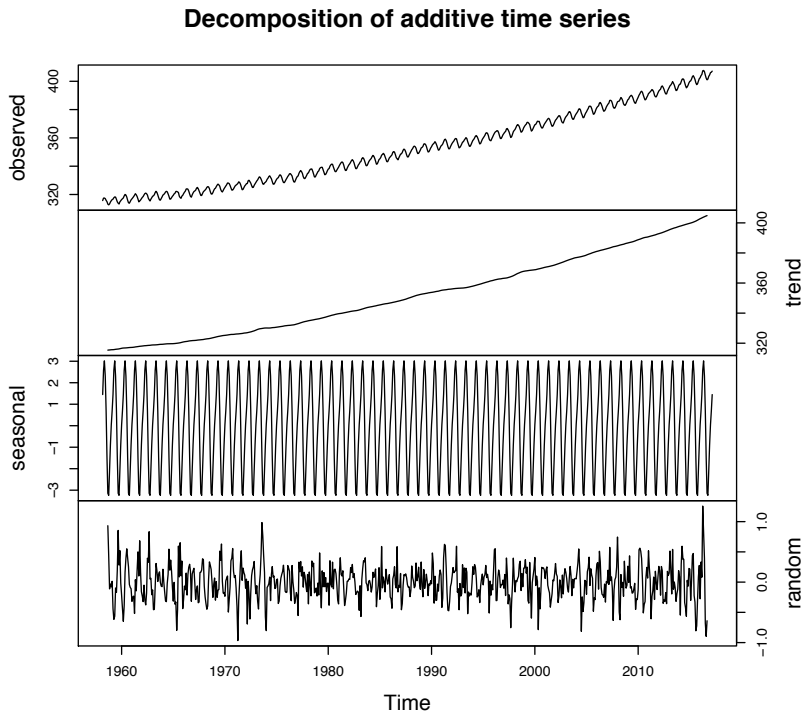


Figure 4.7: Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors obtained with the function `decompose()`.

- ```type``` type of error (```"additive"``` or ```"multiplicative"```)

We can easily make plots of the output and compare them to those in Figure 4.6:

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

The results obtained with `decompose()` (Figure 4.7) 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.

4.3 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

$$\nabla x_t = x_t - x_{t-1}, \tag{4.6}$$

and, more generally, for order d

$$\nabla^d x_t = (1 - \mathbf{B})^d x_t, \tag{4.7}$$

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

$$x_t = x_{t-1} + w_t, \text{ with } w_t \sim \mathrm{N}(0, q). \tag{4.8}$$

Applying the difference operator to Equation (4.8) will yield a time series of Gaussian white noise errors $\{w_t\}$:

$$\begin{aligned} &\nabla(x_t = x_{t-1} + w_t) \\ x_t - x_{t-1} &= x_{t-1} - x_{t-1} + w_t \\ x_t - x_{t-1} &= w_t \end{aligned} \tag{4.9}$$

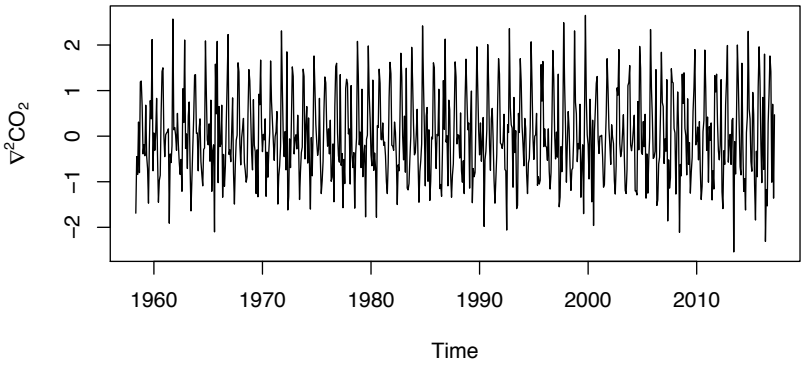


Figure 4.8: Time series of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

4.3.1 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 (4.7)). 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 4.1 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])))
```

We were apparently successful in removing the trend, but the seasonal effect still appears obvious (Figure 4.8). Therefore, let's go ahead and difference that series at lag-12 because our data were collected monthly.

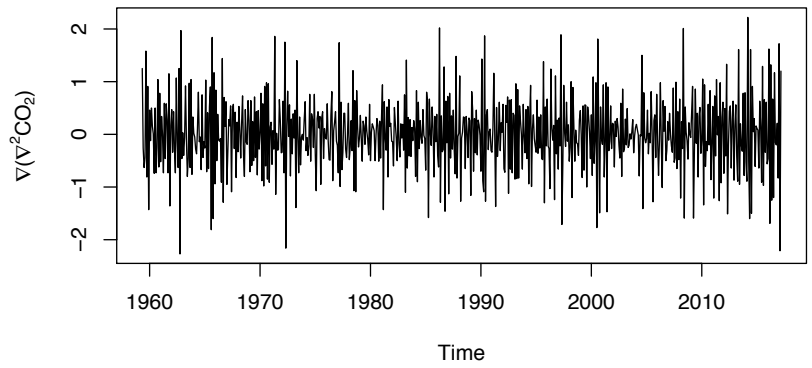


Figure 4.9: Time series of the lag-12 difference of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

```
## 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],")")))
```

Now we have a time series that appears to be random errors without any obvious trend or seasonal components (Figure 4.9).

4.4 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.

4.4.1 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

$$c_k = \frac{1}{n}\sum_{t=1}^{n-k}(x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{4.10}$$

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

$$r_k = \frac{c_k}{c_0} = \text{Cor}(x_t, x_{t+k}) \tag{4.11}$$

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

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{4.12}$$

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 autocovariance (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)
```

There are 4 things about Figure 4.10 that are noteworthy:

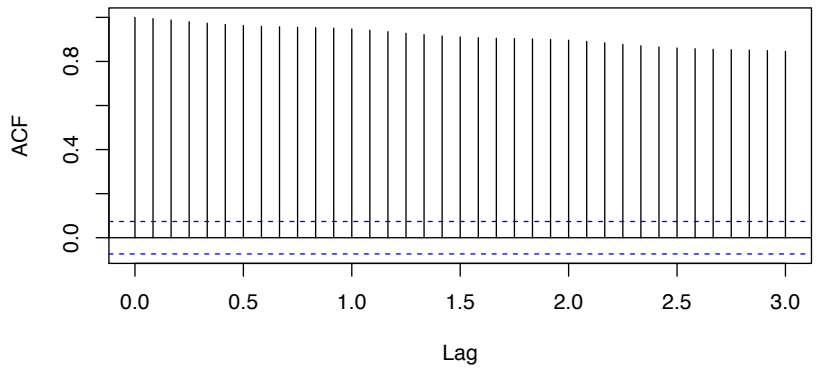


Figure 4.10: Correlogram of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `acf()`.

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 default plots for **acf** objects, let's define a new plot function for **acf** objects with some better features:

```
## better ACF plot
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.

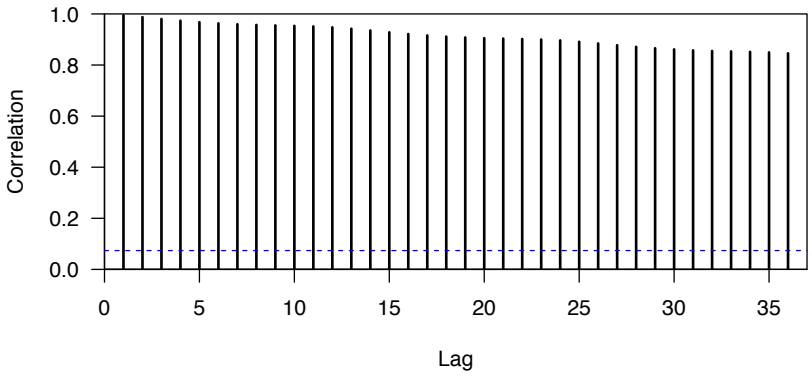
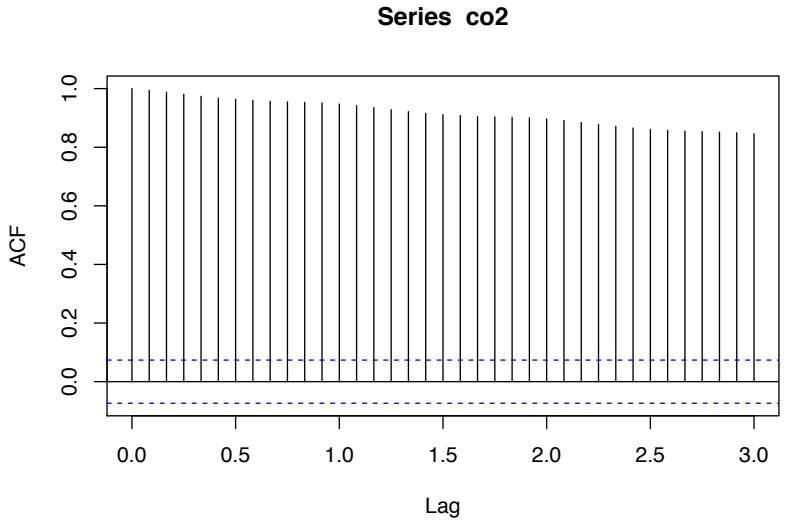


Figure 4.11: Correlogram of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `plot.acf()`.

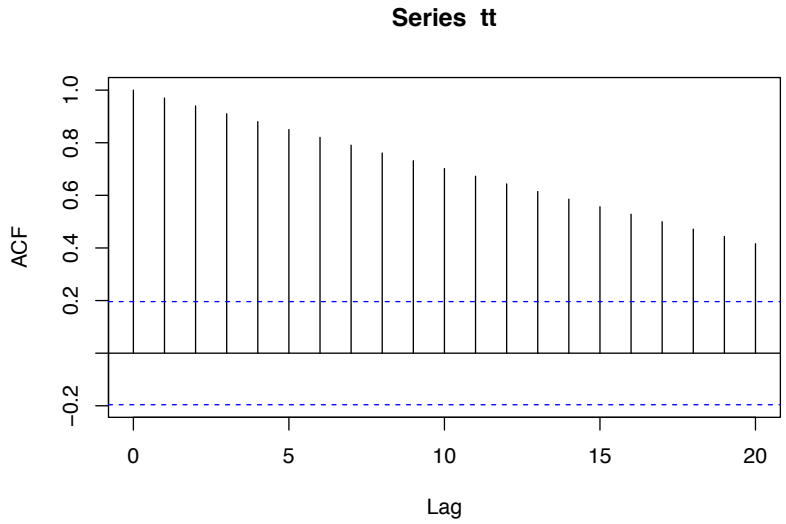
```
## acf of the CO2 data
co2.acf <- acf(co2, lag.max=36)
## correlogram of the CO2 data
plot.acf(co2.acf)
```



Notice that all of the relevant information is still there (Figure 4.11), 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)
```



The correlogram for a straight line is itself a linearly decreasing function over time (Figure 4.12).

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

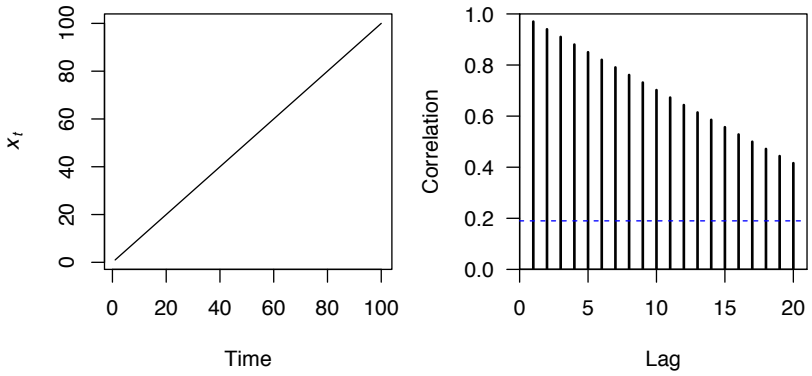


Figure 4.12: Time series plot of a straight line (left) and the correlogram of its ACF (right).

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)
```

Perhaps not surprisingly, the correlogram for a sine wave is itself a sine wave whose amplitude decreases linearly over time (Figure 4.13).

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])))
```

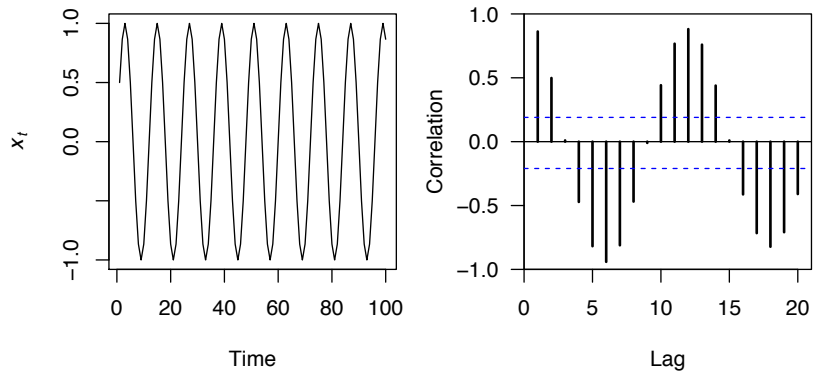


Figure 4.13: Time series plot of a discrete sine wave (left) and the correlogram of its ACF (right).

```
## 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 4.14).

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.

4.4.2 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 $\{x_{t-1}, x_{t-2}, \ldots, x_{t-(k-1)}\}$ removed. Recall from lecture that we define the PACF as

$$f_k = \begin{cases} \text{Cor}(x_1, x_0) = r_1 & \text{if } k = 1; \\ \text{Cor}(x_k - x_k^{k-1}, x_0 - x_0^{k-1}) & \text{if } k \geq 2; \end{cases} \tag{4.13}$$

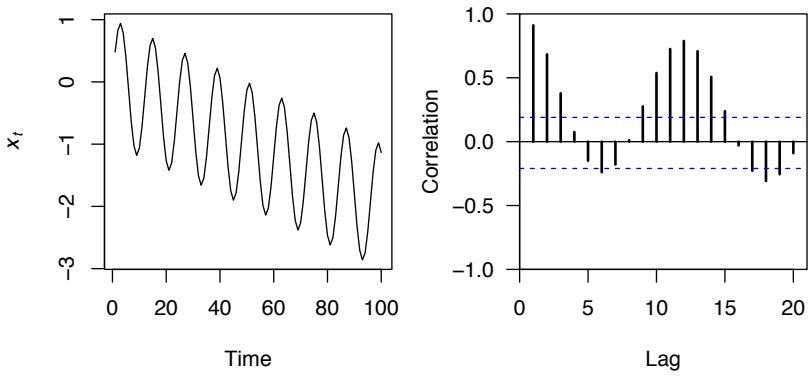


Figure 4.14: Time series plot of a discrete sine wave (left) and the correlogram of its ACF (right).

with

$$x_k^{k-1} = \beta_1 x_{k-1} + \beta_2 x_{k-2} + \cdots + \beta_{k-1} x_1; \tag{4.14a}$$
$$x_0^{k-1} = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_{k-1} x_{k-1}. \tag{4.14b}$$

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.

```
## better PACF plot
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)
```

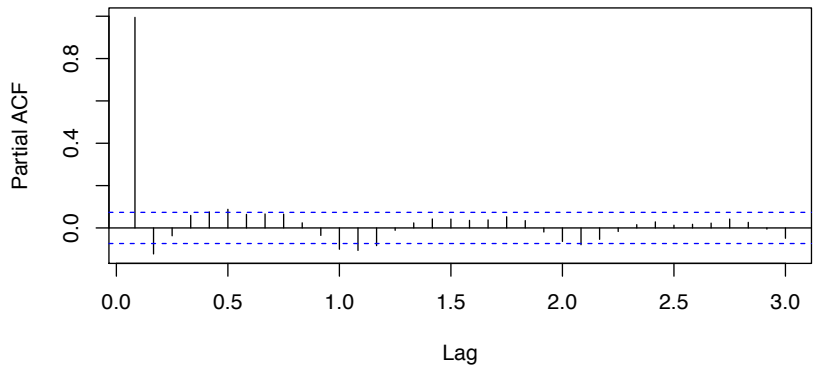


Figure 4.15: Correlogram of the PACF for the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `pacf()`.

```
  abline(h=-1/nn+c(-2,2)/sqrt(nn),lty="dashed",col="blue")
  abline(h=0)
}
```

Notice in Figure 4.15 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.

4.4.3 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

$$g_k^{xy} = \frac{1}{n} \sum_{t=1}^{n-k} (y_t - \bar{y}) (x_{t+k} - \bar{x}) , \tag{4.15}$$

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

$$r_k^{xy} = \frac{g_k^{xy}}{\sqrt{\text{SD}_x \text{SD}_y}}; \tag{4.16}$$

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

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{4.17}$$

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 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 4.16 and 4.17 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).

4.5 White noise (WN)

A time series $\{w_t\}$ is a discrete white noise series (DWN) if the $w_1, w_1, \ldots, w_t$ 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

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

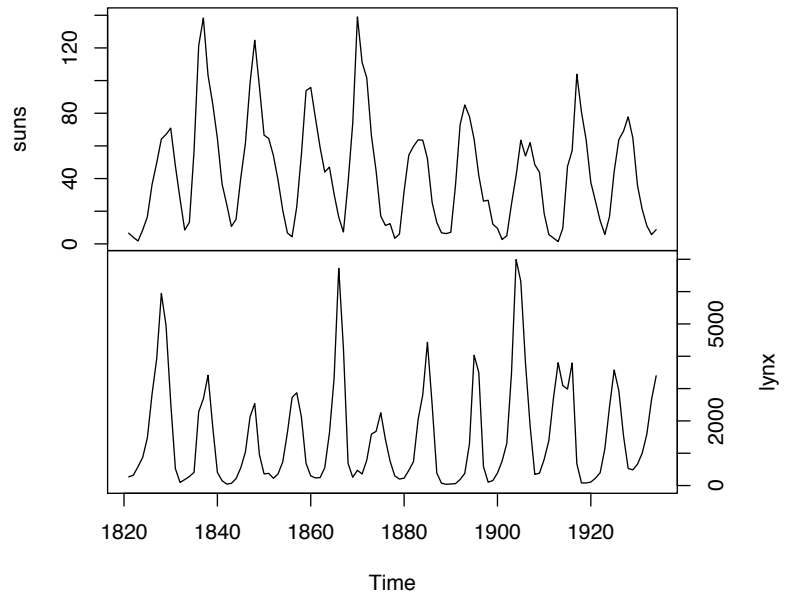


Figure 4.16: Time series of sunspot activity (top) and lynx trappings in Canada (bottom) from 1821-1934.

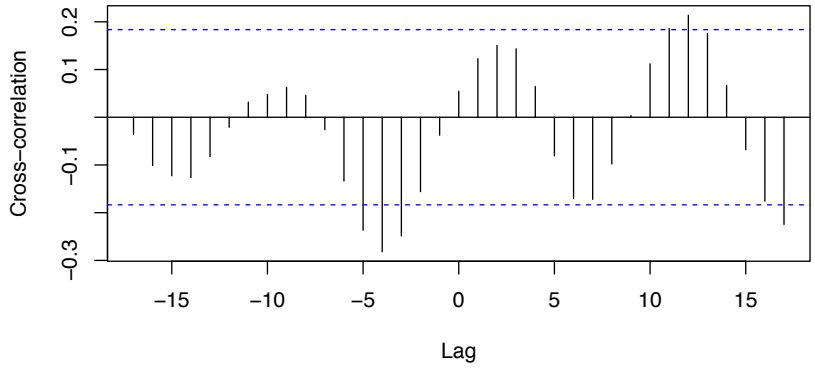


Figure 4.17: CCF for annual sunspot activity and the log of the number of lynx trappings in Canada from 1821-1934.

our time series 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):

$$\begin{aligned} \bar{x} &= 0 \\ c_k &= \text{Cov}(e_t, e_{t+k}) = \begin{cases} q & \text{if } k = 0 \\ 0 & \text{if } k \neq 1 \end{cases} \\ r_k &= \text{Cor}(e_t, e_{t+k}) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k \neq 1. \end{cases} \end{aligned} \tag{4.18}$$

4.5.1 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 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

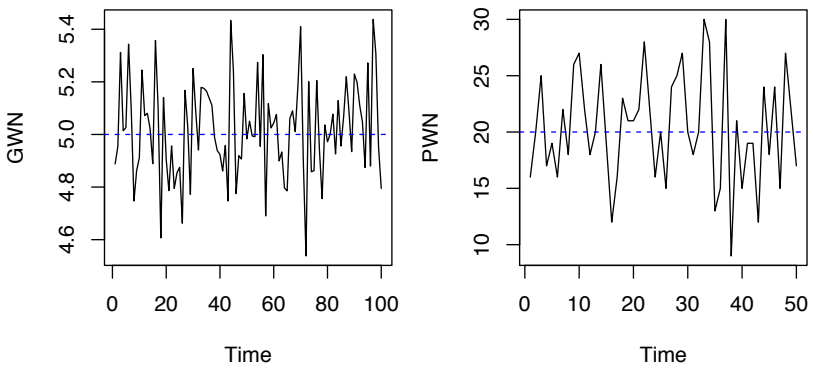


Figure 4.18: Time series plots of simulated Gaussian (left) and Poisson (right) white noise.

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")
```

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 are dealing with a sample of the distributions rather than the entire population

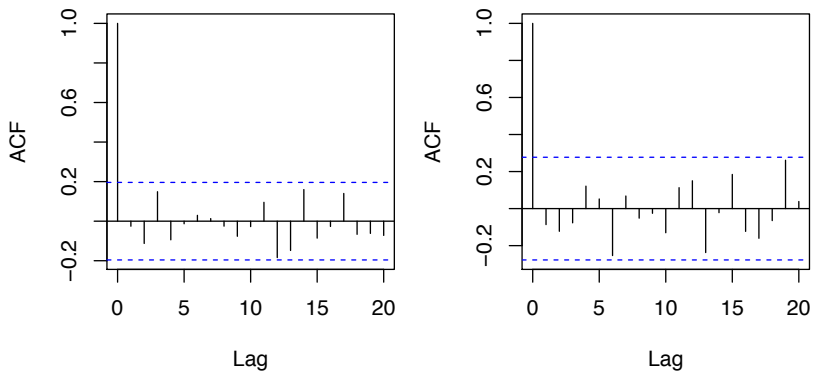


Figure 4.19: ACF's for the simulated Gaussian (left) and Poisson (right) white noise shown in Figure 4.18.

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.

4.6 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

$$x_t = x_{t-1} + w_t, \tag{4.19}$$

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.

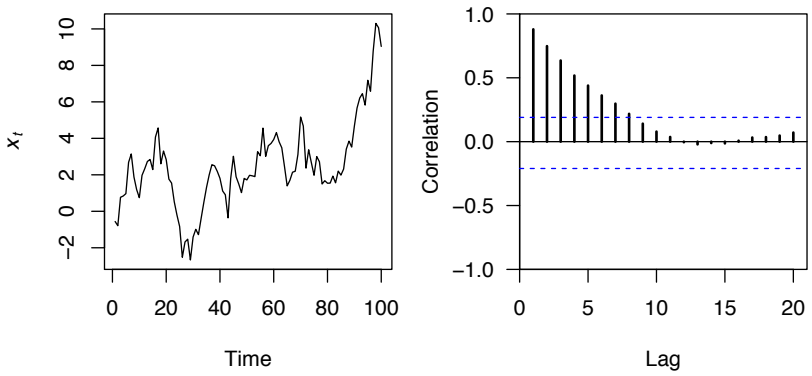


Figure 4.20: Simulated time series of a random walk model (left) and its associated ACF (right).

4.6.1 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))
```

Perhaps not surprisingly based on their names, autoregressive models such as RW's have a high degree of autocorrelation out to long lags (Figure 4.20).

4.6.2 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 (4.19), it is easy to see that

$$\begin{aligned} x_t &= x_{t-1} + w_t \\ x_{t-1} &= x_{t-2} + w_{t-1} \\ x_{t-2} &= x_{t-3} + w_{t-2} \\ &\vdots \end{aligned} \tag{4.20}$$

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

$$x_t = w_t + w_{t-1} + w_{t-2} + \cdots + w_{t-\infty} + x_{t-\infty}. \tag{4.21}$$

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

$$\begin{aligned} x_t &= w_1 + w_2 + \cdots + w_t \\ &= \sum_{t=1}^{T} w_t. \end{aligned} \tag{4.22}$$

From Equation (4.22) 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)
```

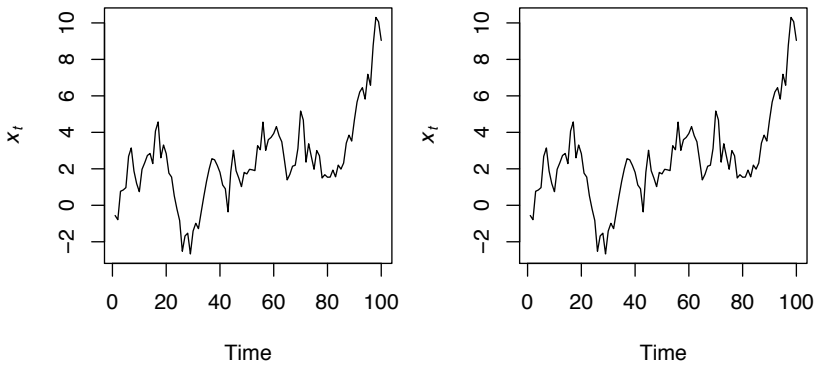


Figure 4.21: Time series of the same random walk model formulated as Equation (4.19) and simulated via a for loop (left), and as Equation (4.22) and simulated via `cumsum()` (right).

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.

4.7 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

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t, \tag{4.23}$$

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2.

For our purposes we usually assume that $w_t \sim \mathrm{N}(0, q)$. Note that the random walk in Equation (4.19) is a special case of an AR(1) model where $\phi_1 = 1$ and $\phi_k = 0$ for $k \geq 2$.

4.7.1 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.

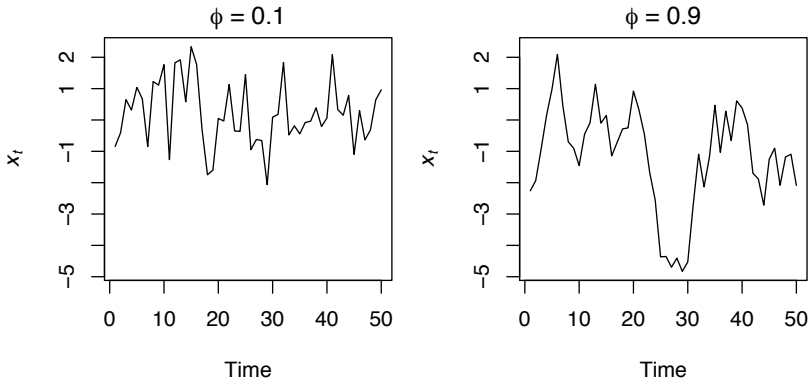


Figure 4.22: Time series of simulated AR(1) processes with $\phi = 0.1$ (left) and $\phi = 0.9$ (right).

```
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 4.22? 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 coefficient, 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)
## 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

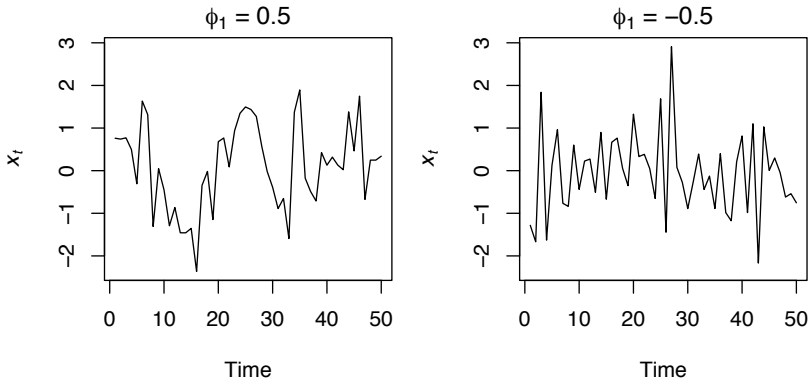


Figure 4.23: Time series of simulated AR(1) processes with $\phi_1 = 0.5$ (left) and $\phi_1 = -0.5$ (right).

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`".

4.7.2 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")
}
```

As we saw in lecture and is evident from our examples shown in Figure 4.24, 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.

4.8 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

$$x_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q w_{t-q}, \tag{4.24}$$

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2; for our purposes we usually assume that $w_t \sim \mathrm{N}(0, q)$. 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

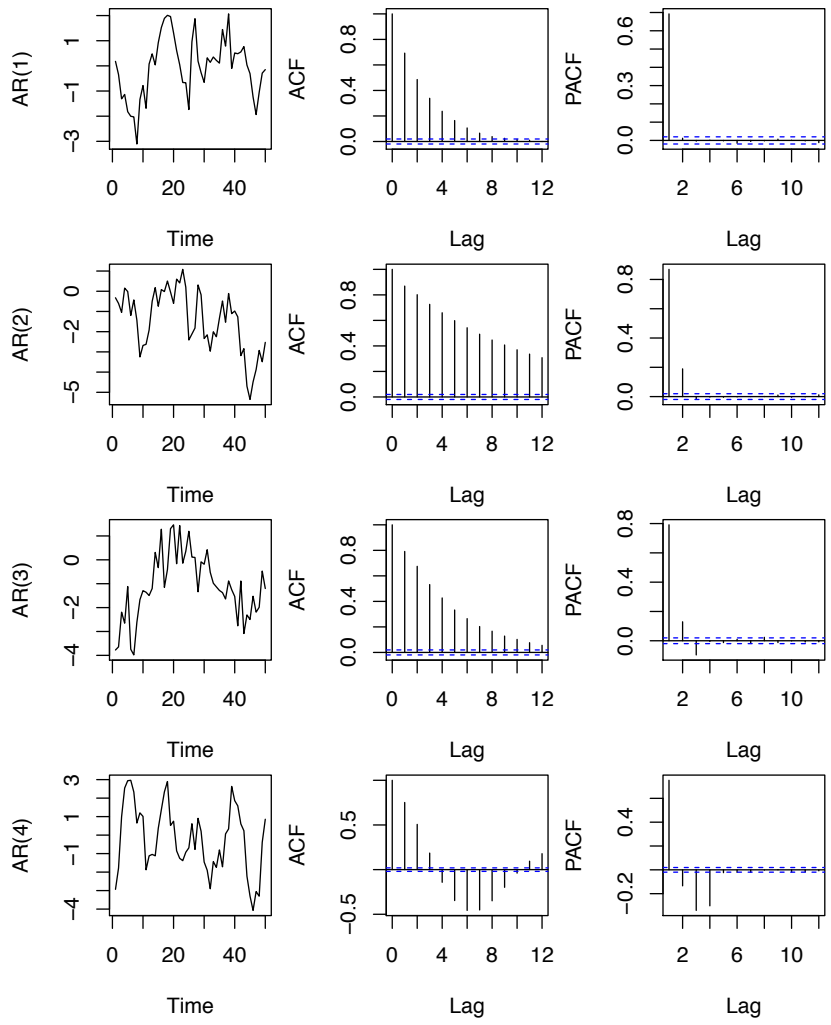


Figure 4.24: Time series of simulated AR(p) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

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

$$\begin{aligned} x_t &= w_t - \theta w_{t-1} \\ x_t &= w_t - \theta \mathbf{B} w_t \\ x_t &= (1 - \theta \mathbf{B}) w_t, \\ &\Downarrow \\ w_t &= \frac{1}{(1 - \theta \mathbf{B})} x_t \\ w_t &= (1 + \theta \mathbf{B} + \theta^2 \mathbf{B}^2 + \theta^3 \mathbf{B}^3 + \dots) x_t \\ w_t &= x_t + \theta x_{t-1} + \theta^2 x_{t-2} + \theta^3 x_{t-3} + \dots \end{aligned} \tag{4.25}$$

4.8.1 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,
```

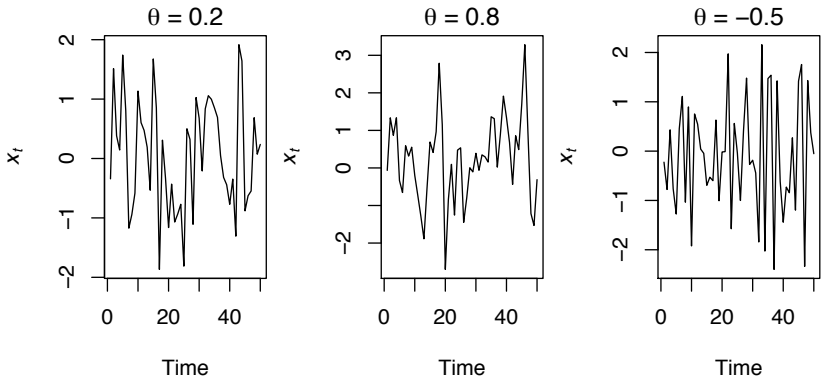


Figure 4.25: Time series of simulated MA(1) processes with $\theta = 0.2$ (left), $\theta = 0.8$ (middle), and $\theta = -0.5$ (right).

```
        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")))
```

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.

4.8.2 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 4.26). 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.

4.9 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

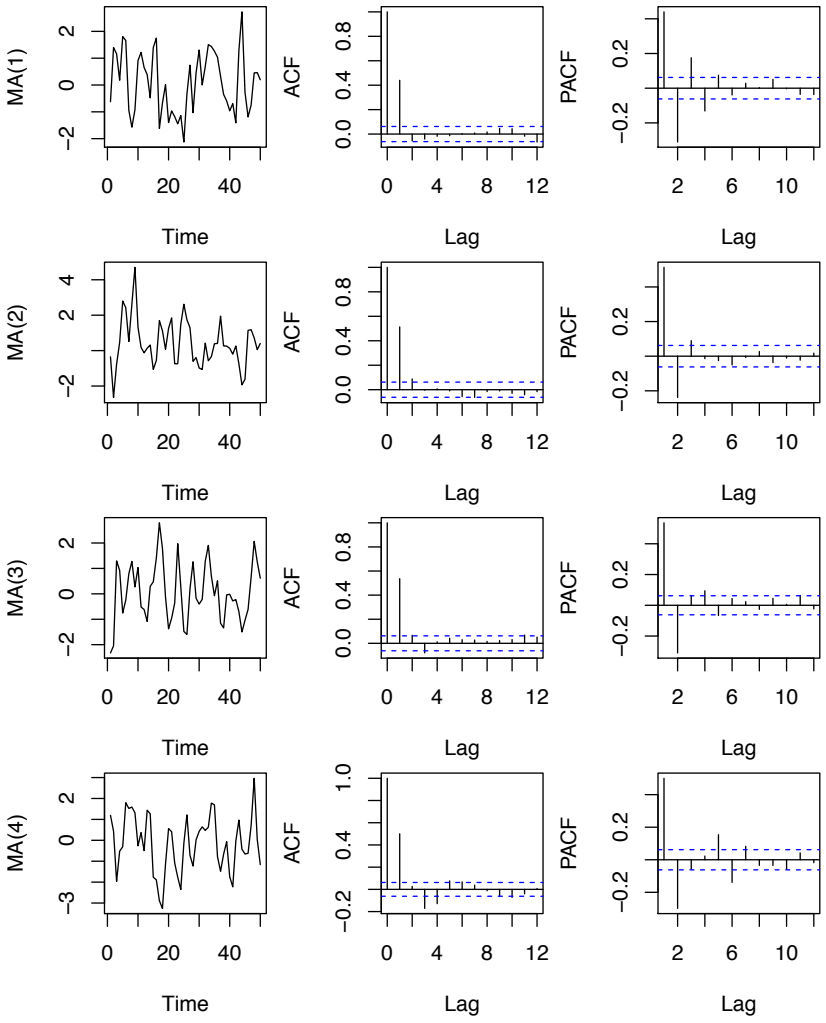


Figure 4.26: Time series of simulated MA(q) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

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

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t + \theta w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q x_{t-q}, \quad (4.26)$$

and the w_t are white noise.

4.9.1 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 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

$$x_t = \mu + \phi(x_{t-1} - \mu) + w_t. \quad (4.27)$$

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 `sigmâ2` 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.

4.9.2 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
  }
}
```

```
Warning in arima(x = ARMA.sim, order = c(p, 0, q)): possible convergence
problem: optim gave code = 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 **forecast** package 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.

```
## 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    mean
      -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`.

4.10 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 colleague 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 load the data using

```
data(hourlyphyto, package="atsalibrary")
pDat <- hourlyphyto
```

Use the information above to do the following:

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)
   ```

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

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.)

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.

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`)

6. Write out the best model in the form of Equation (4.26) 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).

Chapter 5

Box-Jenkins method

In this chapter, you will practice selecting and fitting an ARIMA model to catch data using the Box-Jenkins method. After fitting a model, you will prepare simple forecasts using the **forecast** package.

Data and packages

We will use the catch landings from Greek waters (`greeklandings`) and the Chinook landings (`chinook`) in Washington data sets for this chapter. These datasets are in the **atsalibrary** package on GitHub. Install using the **devtools** package.

```
library(devtools)
devtools::install_github("nwfsc-timeseries/atsalibrary")
```

Load the data.

```
data(greeklandings, package="atsalibrary")
landings <- greeklandings
# Use the monthly data
data(chinook, package="atsalibrary")
chinook <- chinook.month
```

Ensure you have the necessary packages.

```
library(ggplot2)
library(gridExtra)
library(reshape2)
library(tseries)
library(urca)
library(forecast)
```

5.1 Box-Jenkins method

A. Model form selection

1. Evaluate stationarity
2. Selection of the differencing level (d) – to fix stationarity problems
3. Selection of the AR level (p)
4. Selection of the MA level (q)

B. Parameter estimation

C. Model checking

5.2 Stationarity

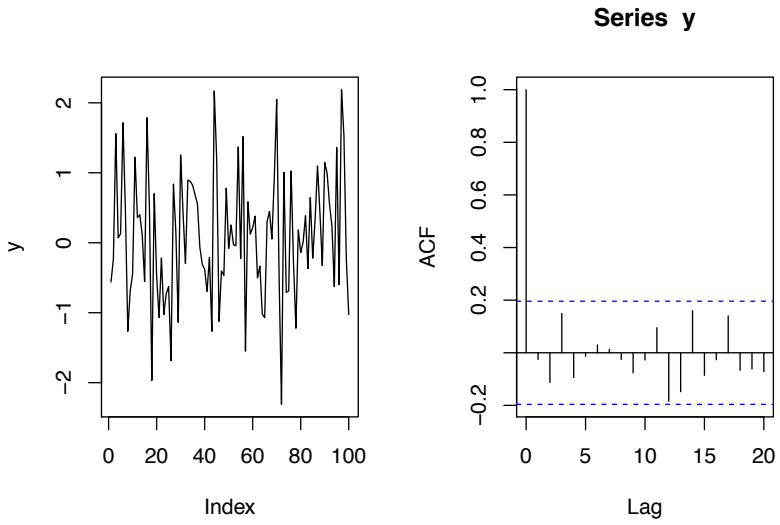
It is important to test and transform (via differencing) your data to ensure stationarity when fitting an ARMA model using standard algorithms. The standard algorithms for ARIMA models assume stationarity and we will be using those algorithms. It possible to fit ARMA models without transforming the data. We will cover that in later chapters. However, that is not commonly done in the literature on forecasting with ARMA models, certainly not in the literature on catch forecasting.

Keep in mind also that many ARMA models are stationary and you do not want to get in the situation of trying to fit an incompatible process model to your data. We will see examples of this when we start fitting models to non-stationary data and random walks.

5.2.1 Look at stationarity in simulated data

We will start by looking at white noise and a stationary AR(1) process from simulated data. White noise is simply a string of random numbers drawn from a Normal distribution. `rnorm()` with return random numbers drawn from a Normal distribution. Use `?rnorm` to understand what the function requires.

```
TT <- 100
y <- rnorm(TT, mean=0, sd=1) # 100 random numbers
op <- par(mfrow=c(1,2))
plot(y, type="l")
acf(y)
```

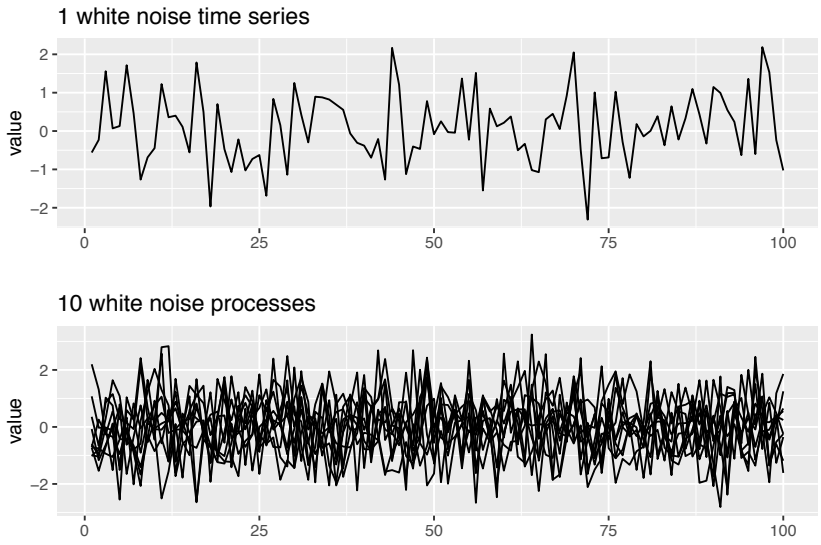



```
par(op)
```

Here we use `ggplot()` to plot 10 white noise time series.

```
dat <- data.frame(t=1:TT, y=y)
p1 <- ggplot(dat, aes(x=t, y=y)) + geom_line() +
  ggtitle("1 white noise time series") + xlab("") + ylab("value")
ys <- matrix(rnorm(TT*10),TT,10)
ys <- data.frame(ys)
```

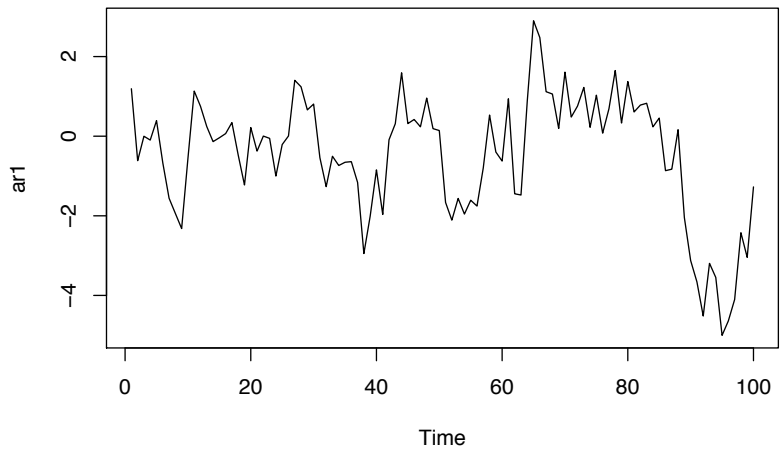
```
ys$id = 1:TT

ys2 <- melt(ys, id.var="id")
p2 <- ggplot(ys2, aes(x=id,y=value,group=variable)) +
  geom_line() + xlab("") + ylab("value") +
  ggtitle("10 white noise processes")
grid.arrange(p1, p2, ncol = 1)
```



These are stationary because the variance and mean (level) does not change with time.

An AR(1) process is also stationary.

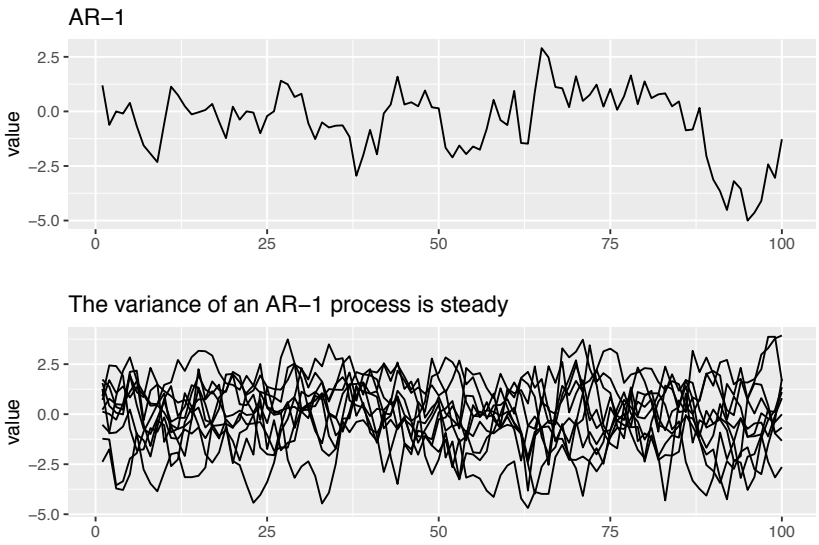
```
theta <- 0.8
nsim <- 10
ar1 <- arima.sim(TT, model=list(ar=theta))
plot(ar1)
```



We can use ggplot to plot 10 AR(1) time series, but we need to change the data to a data frame.

```
dat <- data.frame(t=1:TT, y=ar1)
p1 <- ggplot(dat, aes(x=t, y=y)) + geom_line() +
  ggtitle("AR-1") + xlab("") + ylab("value")
ys <- matrix(0,TT,nsim)
for(i in 1:nsim) ys[,i] <- as.vector(arima.sim(TT, model=list(ar=theta)))
ys <- data.frame(ys)
ys$id <- 1:TT

ys2 <- melt(ys, id.var="id")
p2 <- ggplot(ys2, aes(x=id,y=value,group=variable)) +
  geom_line() + xlab("") + ylab("value") +
  ggtitle("The variance of an AR-1 process is steady")
grid.arrange(p1, p2, ncol = 1)
```

```
Don't know how to automatically pick scale for object of type ts. Defaulting to conti
```



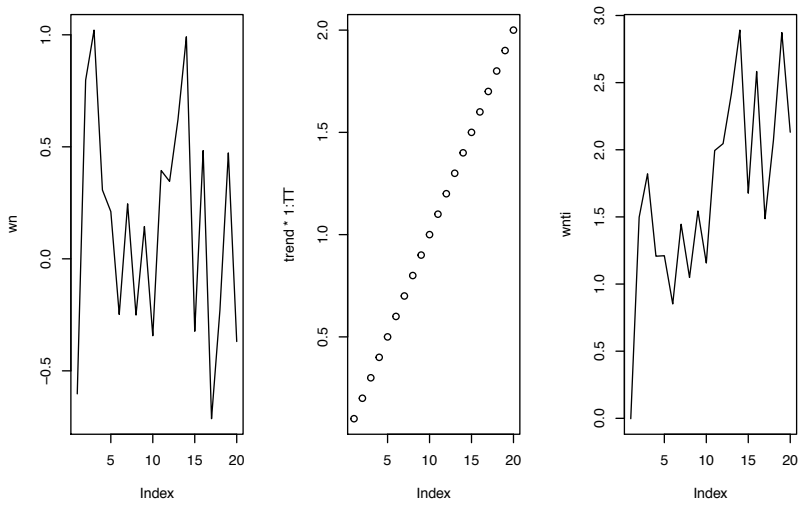
5.2.2 Stationary around a linear trend

Fluctuating around a linear trend is a very common type of stationarity used in ARMA modeling and forecasting. This is just a stationary process, like white noise or AR(1), around an linear trend up or down.

```
intercept <- .5
trend <- 0.1
sd <- 0.5
TT <- 20
wn <- rnorm(TT, sd=sd) #white noise
wni <- wn+intercept #white noise witn interept
wnti <- wn + trend*(1:TT) + intercept
```

See how the white noise with trend is just the white noise overlaid on a linear trend.

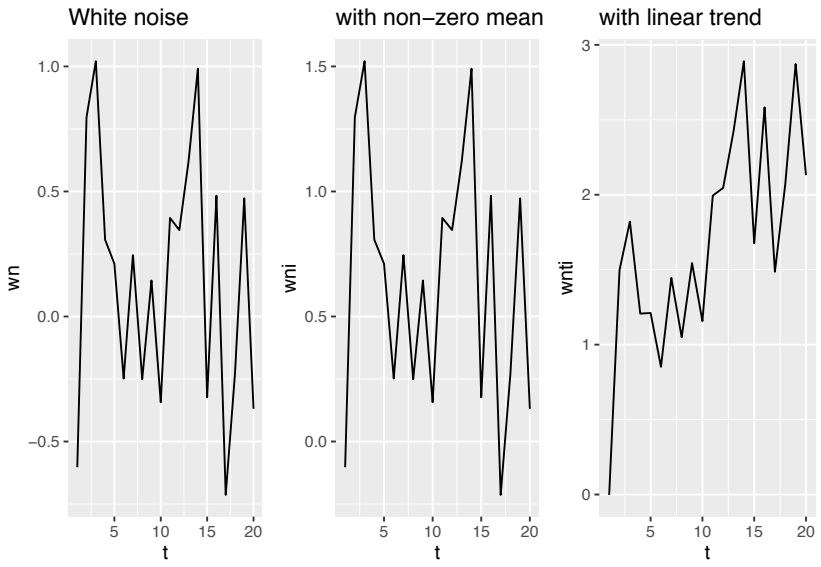
```
op <- par(mfrow=c(1,3))
plot(wn, type="l")
plot(trend*1:TT)
plot(wnti, type="l")
```




```
par(op)
```

We can make a similar plot with ggplot.

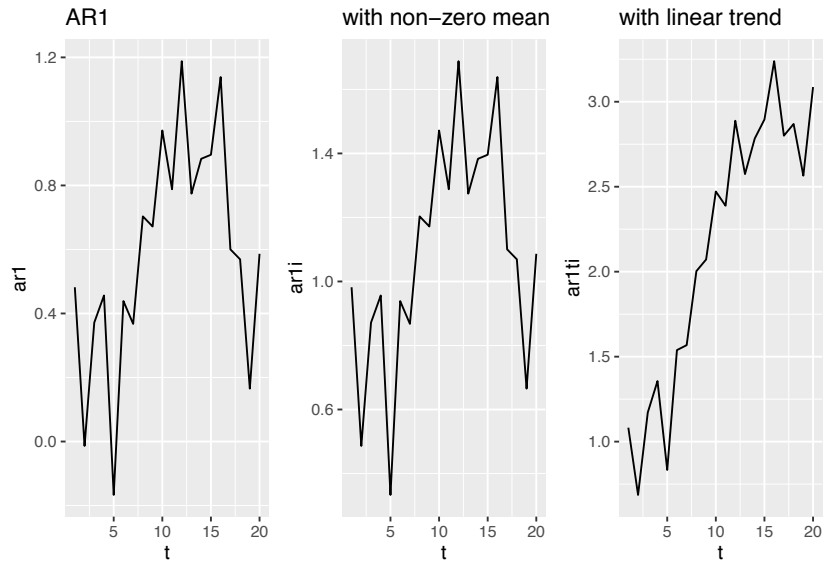
```
dat <- data.frame(t=1:TT, wn=wn, wni=wni, wnti=wnti)
p1 <- ggplot(dat, aes(x=t, y=wn)) + geom_line() + ggtitle("White noise")
p2 <- ggplot(dat, aes(x=t, y=wni)) + geom_line() + ggtitle("with non-zero mean")
p3 <- ggplot(dat, aes(x=t, y=wnti)) + geom_line() + ggtitle("with linear trend")
grid.arrange(p1, p2, p3, ncol = 3)
```



We can make a similar plot with AR(1) data. Ignore the warnings about not knowing how to pick the scale.

```
beta1 <- 0.8
ar1 <- arima.sim(TT, model=list(ar=beta1), sd=sd)
ar1i <- ar1 + intercept
ar1ti <- ar1 + trend*(1:TT) + intercept
dat <- data.frame(t=1:TT, ar1=ar1, ar1i=ar1i, ar1ti=ar1ti)
p4 <- ggplot(dat, aes(x=t, y=ar1)) + geom_line() + ggtitle("AR1")
p5 <- ggplot(dat, aes(x=t, y=ar1i)) + geom_line() + ggtitle("with non-zero me
p6 <- ggplot(dat, aes(x=t, y=ar1ti)) + geom_line() + ggtitle("with linear tre

grid.arrange(p4, p5, p6, ncol = 3)
```

```
Don't know how to automatically pick scale for object of type ts. Defaulting t
Don't know how to automatically pick scale for object of type ts. Defaulting t
Don't know how to automatically pick scale for object of type ts. Defaulting t
```



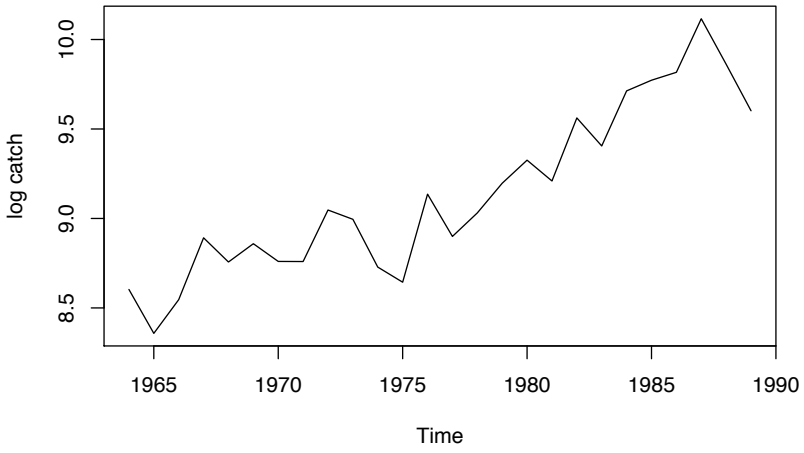
5.2.3 Greek landing data

We will look at the anchovy data. Notice the two == in the subset call not one =. We will use the Greek data before 1989 for the lab.

```
anchovy <- subset(landings, Species=="Anchovy" & Year <= 1989)$log.metric.tons
anchovyts <- ts(anchovy, start=1964)
```

Plot the data.

```
plot(anchovyts, ylab="log catch")
```



Questions to ask.

- Does it have a trend (goes up or down)? Yes, definitely
- Does it have a non-zero mean? Yes
- Does it look like it might be stationary around a trend? Maybe

5.3 Dickey-Fuller and Augmented Dickey-Fuller tests

5.3.1 Dickey-Fuller test

The Dickey-Fuller test is testing if $\phi = 0$ in this model of the data:

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

which is written as

$$\Delta y_t = y_t - y_{t-1} = \alpha + \beta t + \gamma y_{t-1} + e_t$$

where y_t is your data. It is written this way so we can do a linear regression of Δy_t against t and y_{t-1} and test if γ is different from 0. If $\gamma = 0$, then we have a random walk process. If not and $-1 < 1 + \gamma < 1$, then we have a stationary process.

5.3.2 Augmented Dickey-Fuller test

The Augmented Dickey-Fuller test allows for higher-order autoregressive processes by including Δy_{t-p} in the model. But our test is still if $\gamma = 0$.

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \delta_2 \Delta y_{t-2} + \dots$$

The null hypothesis for both tests is that the data are non-stationary. We want to REJECT the null hypothesis for this test, so we want a p-value of less that 0.05 (or smaller).

5.3.3 ADF test using `adf.test()`

The `adf.test()` from the **tseries** package will do a Augmented Dickey-Fuller test (Dickey-Fuller if we set lags equal to 0) with a trend and an intercept. Use `?adf.test` to read about this function. The function is

```
adf.test(x, alternative = c("stationary", "explosive"),
         k = trunc((length(x)-1)^(1/3)))
```

`x` are your data. `alternative="stationary"` means that $-2 < \gamma < 0$ ($-1 < \phi < 1$) and `alternative="explosive"` means that is outside these bounds. `k` is the number of δ lags. For a Dickey-Fuller test, so only up to AR(1) time dependency in our stationary process, we set `k=0` so we have no δ's in our test. Being able to control the lags in our test, allows us to avoid a stationarity test that is too complex to be supported by our data.

5.3.3.1 Test on white noise

Let's start by doing the test on data that we know are stationary, white noise. We will use an Augmented Dickey-Fuller test where we use the default number of lags (amount of time-dependency) in our test. For a time-series of 100, this is 4.

```
TT <- 100
wn <- rnorm(TT) # white noise
tseries::adf.test(wn)
```

```
Warning in tseries::adf.test(wn): p-value smaller than printed p-value
```

```

    Augmented Dickey-Fuller Test

data:  wn
Dickey-Fuller = -4.8309, Lag order = 4, p-value = 0.01
alternative hypothesis: stationary
```

The null hypothesis is rejected.

Try a Dickey-Fuller test. This is testing with a null hypothesis of AR(1) stationarity versus a null hypothesis with AR(4) stationarity when we used the default `k`.

```
tseries::adf.test(wn, k=0)
```

```
Warning in tseries::adf.test(wn, k = 0): p-value smaller than printed p-value
```

```

    Augmented Dickey-Fuller Test

data:  wn
Dickey-Fuller = -10.122, Lag order = 0, p-value = 0.01
alternative hypothesis: stationary
```

Notice that the test-statistic is smaller. This is a more restrictive test and we can reject the null with a higher significance level.

5.3.3.2 Test on white noise with trend

Try the test on white noise with a trend and intercept.

```
intercept <- 1
wnt <- wn + 1:TT + intercept
tseries::adf.test(wnt)
```

```
Warning in tseries::adf.test(wnt): p-value smaller than printed p-value
```

```

    Augmented Dickey-Fuller Test
```

```
data:  wnt
Dickey-Fuller = -4.8309, Lag order = 4, p-value = 0.01
alternative hypothesis: stationary
```

The null hypothesis is still rejected. `adf.test()` uses a model that allows an intercept and trend.

5.3.3.3 Test on random walk

Let's try the test on a random walk (nonstationary).

```
rw <- cumsum(rnorm(TT))
tseries::adf.test(rw)
```

```
    Augmented Dickey-Fuller Test

data:  rw
Dickey-Fuller = -2.3038, Lag order = 4, p-value = 0.4508
alternative hypothesis: stationary
```

The null hypothesis is NOT rejected as the p-value is greater than 0.05.

Try a Dickey-Fuller test.

```
tseries::adf.test(rw, k=0)
```

```
    Augmented Dickey-Fuller Test

data:  rw
Dickey-Fuller = -1.7921, Lag order = 0, p-value = 0.6627
alternative hypothesis: stationary
```

Notice that the test-statistic is larger.

5.3.3.4 Test the anchovy data

```
tseries::adf.test(anchovyts)
```

```
    Augmented Dickey-Fuller Test

data:  anchovyts
Dickey-Fuller = -1.6851, Lag order = 2, p-value = 0.6923
alternative hypothesis: stationary
```

The p-value is greater than 0.05. We cannot reject the null hypothesis. The null hypothesis is that the data are non-stationary.

5.3.4 ADF test using `ur.df()`

The `ur.df()` Augmented Dickey-Fuller test in the **urca** package gives us a bit more information on and control over the test.

```
ur.df(y, type = c("none", "drift", "trend"), lags = 1,
      selectlags = c("Fixed", "AIC", "BIC"))
```

The `ur.df()` function allows us to specify whether to test stationarity around a zero-mean with no trend, around a non-zero mean with no trend, or around a trend with an intercept. This can be useful when we know that our data have no trend, for example if you have removed the trend already. `ur.df()` allows us to specify the lags or select them using model selection.

5.3.4.1 Test on white noise

Let's first do the test on data we know is stationary, white noise. We have to choose the `type` and `lags`. If you have no particular reason to not include an intercept and trend, then use `type="trend"`. This allows both intercept and trend. When you might you have a particular reason not to use `"trend"`? When you have removed the trend and/or intercept.

Next you need to chose the `lags`. We will use `lags=0` to do the Dickey-Fuller test. Note the number of lags you can test will depend on the amount of data that you have. `adf.test()` used a default of `trunc((length(x)-1)^(1/3))` for the lags, but `ur.df()` requires that you pass in a value or use a fixed default of 1.

`lags=0` is fitting this model to the data. You are testing if the effect for `z.lag.1` is 0.

`z.diff = gamma * z.lag.1 + intercept + trend * tt` `z.diff` means Δy_t and `z.lag.1` is y_{t-1}.

When you use `summary()` for the output from `ur.df()`, you will see the estimated values for γ (denoted `z.lag.1`), intercept and trend. If you see `***` or `**` on the coefficients list for `z.lag.1`, it indicates that the effect of `z.lag.1` is significantly different than 0 and this supports the assumption of stationarity.

The `intercept` and `tt` estimates indicate where there is a non-zero level (intercept) or linear trend (tt).

```
wn <- rnorm(TT)
test <- urca::ur.df(wn, type="trend", lags=0)
summary(test)
```

```
###############################################
# Augmented Dickey-Fuller Test Unit Root Test #
###############################################

Test regression trend


Call:
lm(formula = z.diff ~ z.lag.1 + 1 + tt)

Residuals:
    Min      1Q  Median      3Q     Max
-2.2170 -0.6654 -0.1210  0.5311  2.6277

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.0776865  0.2037709   0.381    0.704
z.lag.1     -1.0797598  0.1014244 -10.646   <2e-16 ***
tt           0.0004891  0.0035321   0.138    0.890
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.004 on 96 degrees of freedom
Multiple R-squared:  0.5416,    Adjusted R-squared:  0.532
F-statistic: 56.71 on 2 and 96 DF,  p-value: < 2.2e-16


Value of test-statistic is: -10.646 37.806 56.7083

Critical values for test statistics:
      1pct  5pct 10pct
tau3 -4.04 -3.45 -3.15
phi2  6.50  4.88  4.16
phi3  8.73  6.49  5.47
```

The coefficient part of the summary indicates that `z.lag.1` is different than 0 (so stationary) and no support for intercept or trend.

Notice that the test statistic is LESS than the critical value for `tau3` at 5 percent. This means the null hypothesis is rejected at $\alpha = 0.05$, a standard level for significance testing.

5.3.4.2 When you might want to use `ur.df()`

If you remove the trend (and/or level) from your data, the `ur.df()` test allows you to increase the power of the test by removing the trend and/or level from the model.

5.4 KPSS test

The null hypothesis for the KPSS test is that the data are stationary. For this test, we do NOT want to reject the null hypothesis. In other words, we want the p-value to be greater than 0.05 not less than 0.05.

5.4.1 Test on simulated data

Let's try the KPSS test on white noise with a trend. The default is a null hypothesis with no trend. We will change this to `null="Trend"`.

```
tseries::kpss.test(wnt, null="Trend")
```

```
Warning in tseries::kpss.test(wnt, null = "Trend"): p-value greater than printed
p-value
```

```

    KPSS Test for Trend Stationarity

data:  wnt
KPSS Trend = 0.045579, Truncation lag parameter = 4, p-value = 0.1
```

The p-value is greater than 0.05. The null hypothesis of stationarity around a trend is not rejected.

Let's try the KPSS test on white noise with a trend but let's use the default of stationary with no trend.

```
tseries::kpss.test(wnt, null="Level")
```

```
Warning in tseries::kpss.test(wnt, null = "Level"): p-value smaller than printed
p-value
```

```

    KPSS Test for Level Stationarity

data:  wnt
KPSS Level = 2.1029, Truncation lag parameter = 4, p-value = 0.01
```

The p-value is less than 0.05. The null hypothesis of stationarity around a level is rejected. This is white noise around a trend so it is definitely a stationary process but has a trend. This illustrates that you need to be thoughtful when applying stationarity tests.

5.4.2 Test the anchovy data

Let's try the anchovy data.

```
kpss.test(anchovyts, null="Trend")
```

```
    KPSS Test for Trend Stationarity

data:  anchovyts
KPSS Trend = 0.14779, Truncation lag parameter = 2, p-value = 0.04851
```

The null is rejected (p-value less than 0.05). Again stationarity is not supported.

5.5 Dealing with non-stationarity

The anchovy data have failed both tests for the stationarity, the Augmented Dickey-Fuller and the KPSS test. How do we fix this? The approach in the Box-Jenkins method is to use differencing.

Let's see how this works with random walk data. A random walk is non-stationary but the difference is white noise so is stationary:

$$x_t - x_{t-1} = e_t, e_t \sim N(0, \sigma)$$

```
adf.test(diff(rw))
```

```
    Augmented Dickey-Fuller Test

data:  diff(rw)
Dickey-Fuller = -3.8711, Lag order = 4, p-value = 0.01834
alternative hypothesis: stationary
```

```
kpss.test(diff(rw))
```

```
Warning in kpss.test(diff(rw)): p-value greater than printed p-value

    KPSS Test for Level Stationarity
```

```
data:  diff(rw)
KPSS Level = 0.30489, Truncation lag parameter = 3, p-value = 0.1
```

If we difference random walk data, the null is rejected for the ADF test and not rejected for the KPSS test. This is what we want.

Let's try a single difference with the anchovy data. A single difference means `dat(t)-dat(t-1)`. We get this using `diff(anchovyts)`.

```
diff1dat <- diff(anchovyts)
adf.test(diff1dat)
```

```

    Augmented Dickey-Fuller Test

data:  diff1dat
Dickey-Fuller = -3.2718, Lag order = 2, p-value = 0.09558
alternative hypothesis: stationary
```

```
kpss.test(diff1dat)
```

```
Warning in kpss.test(diff1dat): p-value greater than printed p-value
```

```

    KPSS Test for Level Stationarity

data:  diff1dat
KPSS Level = 0.089671, Truncation lag parameter = 2, p-value = 0.1
```

If a first difference were not enough, we would try a second difference which is the difference of a first difference.

```
diff2dat <- diff(diff1dat)
adf.test(diff2dat)
```

```
Warning in adf.test(diff2dat): p-value smaller than printed p-value
```

```

    Augmented Dickey-Fuller Test

data:  diff2dat
Dickey-Fuller = -4.8234, Lag order = 2, p-value = 0.01
```

```
alternative hypothesis: stationary
```

The null hypothesis of a random walk is now rejected so you might think that a 2nd difference is needed for the anchovy data. However the actual problem is that the default for `adf.test()` includes a trend but we removed the trend with our first difference. Thus we included an unneeded trend parameter in our test. Our data are not that long and this affects the result.

Let's repeat without the trend and we'll see that the null hypothesis is rejected. The number of lags is set to be what would be used by `adf.test()`. See `?adf.test`.

```
k <- trunc((length(diff1dat)-1)^(1/3))
test <- urca::ur.df(diff1dat, type="drift", lags=k)
summary(test)
```

```
###############################################
# Augmented Dickey-Fuller Test Unit Root Test #
###############################################

Test regression drift


Call:
lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)

Residuals:
     Min       1Q   Median       3Q      Max
-0.37551 -0.13887  0.04753  0.13277  0.28223

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.11062    0.06165   1.794  0.08959 .
z.lag.1     -2.16711    0.64900  -3.339  0.00365 **
z.diff.lag1  0.58837    0.47474   1.239  0.23113
z.diff.lag2  0.13273    0.25299   0.525  0.60623
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 0.207 on 18 degrees of freedom
Multiple R-squared:  0.7231,    Adjusted R-squared:  0.677
F-statistic: 15.67 on 3 and 18 DF,  p-value: 2.918e-05


Value of test-statistic is: -3.3391 5.848

Critical values for test statistics:
      1pct  5pct 10pct
tau2 -3.75 -3.00 -2.63
phi1  7.88  5.18  4.12
```

5.5.1 ndiffs()

As an alternative to trying many different differences and remembering to include or not include the trend or level, you can use the `ndiffs()` function in the **forecast** package. This automates finding the number of differences needed.

```
forecast::ndiffs(anchovyts, test="kpss")
```

```
[1] 1
```

```
forecast::ndiffs(anchovyts, test="adf")
```

```
[1] 1
```

One difference is required to pass both the ADF and KPSS stationarity tests.

5.6 Summary: stationarity testing

The basic stationarity diagnostics are the following

- Plot your data. Look for
  - An increasing trend
  - A non-zero level (if no trend)

  - Strange shocks or steps in your data (indicating something dramatic changed like the data collection methodology)
- Apply stationarity tests
  - `adf.test()` p-value should be less than 0.05 (reject null)
  - `kpss.test()` p-value should be greater than 0.05 (do not reject null)
- If stationarity tests are failed, then try differencing to correct
  - Try `ndiffs()` in the **forecast** package or manually try different differences.

5.7 Estimating ARMA parameters

Let's start with fitting to simulated data.

5.7.1 AR(2) data

Simulate AR(2) data and add a mean level so that the data are not mean 0.

$$x_t = 0.8x_{t-1} + 0.1x_{t-2} + e_t y_t = x_t + m$$

```
m <- 1
ar2 <- arima.sim(n=1000, model=list(ar=c(.8,.1))) + m
```

To see info on `arima.sim()`, type `?arima.sim`.

5.7.2 Fit with `Arima()`

Fit an ARMA(2) with level to the data.

```
forecast::Arima(ar2, order=c(2,0,0), include.constant=TRUE)
```

```
Series: ar2
ARIMA(2,0,0) with non-zero mean

Coefficients:
```

```
         ar1     ar2    mean
      0.7684  0.1387  0.9561
s.e.  0.0314  0.0314  0.3332

sigma^2 estimated as 0.9832:  log likelihood=-1409.77
AIC=2827.54   AICc=2827.58   BIC=2847.17
```

Note, the model being fit by `Arima()` is not this model

$$y_t = m + 0.8y_{t-1} + 0.1y_{t-2} + e_t$$

It is this model:

$$(y_t - m) = 0.8(y_{t-1} - m) + 0.1(y_{t-2} - m) + e_t$$

or as written above:

$$x_t = 0.8x_{t-1} + 0.1x_{t-2} + e_t y_t = x_t + m$$

We could also use `arima()` to fit to the data.

```
arima(ar2, order=c(2,0,0), include.mean=TRUE)
```

```
Warning in arima(ar2, order = c(2, 0, 0), include.mean = TRUE): possible
convergence problem: optim gave code = 1


Call:
arima(x = ar2, order = c(2, 0, 0), include.mean = TRUE)

Coefficients:
         ar1     ar2  intercept
      0.7684  0.1387     0.9561
s.e.  0.0314  0.0314     0.3332

sigma^2 estimated as 0.9802:  log likelihood = -1409.77,  aic = 2827.54
```

However we will not be using `arima()` directly because for if we have differenced data, it will not allow us to include and estimated mean level. Unless we have transformed our differenced data in a way that ensures it is mean zero, then we want to include a mean.

Try increasing the length of the simulated data (from 100 to 1000 say) and see how that affects your parameter estimates. Run the simulation a few times.

5.7.3 AR(1) simulated data

```
ar1 <- arima.sim(n=100, model=list(ar=c(.8)))+m
forecast::Arima(ar1, order=c(1,0,0), include.constant=TRUE)
```

```
Series: ar1
ARIMA(1,0,0) with non-zero mean

Coefficients:
         ar1    mean
      0.7091  0.4827
s.e.  0.0705  0.3847

sigma^2 estimated as 1.34:  log likelihood=-155.85
AIC=317.7   AICc=317.95   BIC=325.51
```

5.7.4 ARMA(1,2) simulated data

Simulate ARMA(1,2)

$$x_t = 0.8x_{t-1} + e_t + 0.8e_{t-1} + 0.2e_{t-2}$$

```
arma12 = arima.sim(n=100, model=list(ar=c(0.8), ma=c(0.8, 0.2)))+m
forecast::Arima(arma12, order=c(1,0,2), include.constant=TRUE)
```

```
Series: arma12
ARIMA(1,0,2) with non-zero mean

Coefficients:
         ar1     ma1     ma2    mean
      0.8138  0.8599  0.1861  0.3350
s.e.  0.0646  0.1099  0.1050  0.8145
```

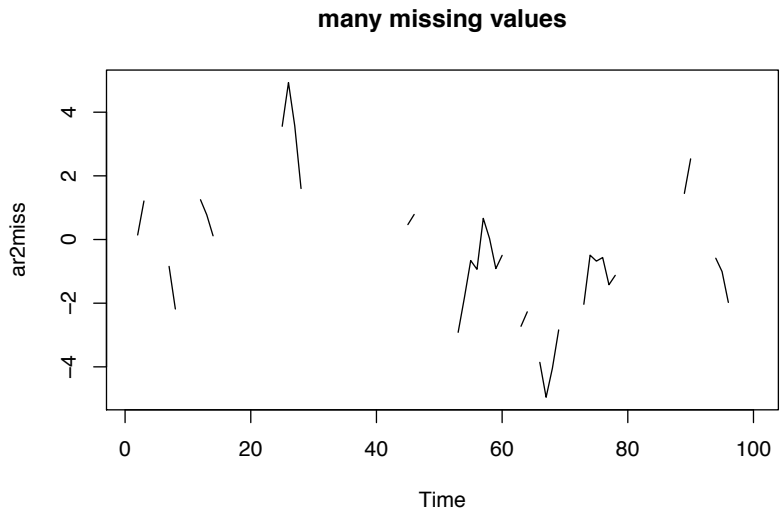
```
sigma^2 estimated as 0.6264:  log likelihood=-118.02
AIC=246.03   AICc=246.67   BIC=259.06
```

We will up the number of data points to 1000 because models with a MA component take a lot of data to estimate. Models with MA(>1) are not very practical for fisheries data for that reason.

5.7.5 These functions work for data with missing values

Create some AR(2) data and then add missing values (NA).

```
ar2miss <- arima.sim(n=100, model=list(ar=c(.8,.1)))
ar2miss[sample(100,50)] <- NA
plot(ar2miss, type="l")
title("many missing values")
```



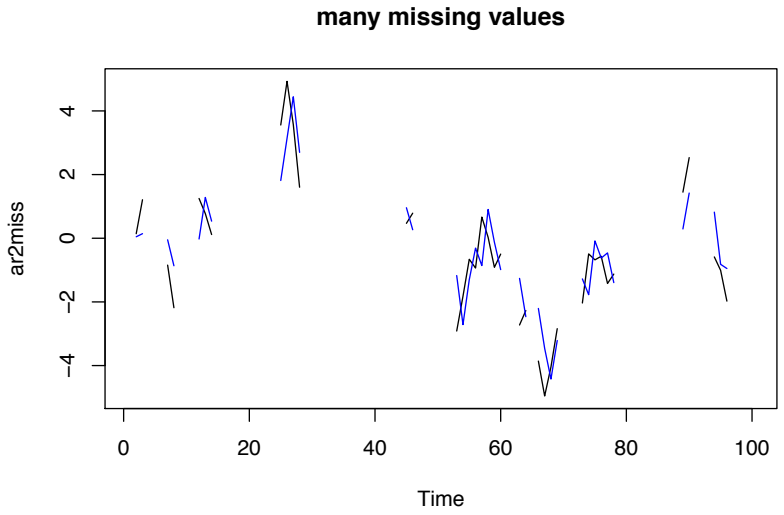
Fit

```
fit <- forecast::Arima(ar2miss, order=c(2,0,0))
fit
```

```
Series: ar2miss
ARIMA(2,0,0) with non-zero mean

Coefficients:
         ar1      ar2     mean
      1.0625  -0.2203  -0.0586
s.e.  0.1555   0.1618   0.6061

sigma^2 estimated as 0.9679:  log likelihood=-79.86
AIC=167.72   AICc=168.15   BIC=178.06
```

Note `fitted()` does not return the expected value at time t. It is the expected value of y_t given the data up to time $t-1$.

```
plot(ar2miss, type="l")
title("many missing values")
lines(fitted(fit), col="blue")
```



It is easy enough to get the expected value of y_t for all the missing values but we'll learn to do that when we learn the **MARSS** package and can apply the Kalman Smoother in that package.

5.8 Estimating the ARMA orders

We will use the `auto.arima()` function in **forecast**. This function will estimate the level of differencing needed to make our data stationary and estimate the AR and MA orders using AICc (or BIC if we choose).

5.8.1 Example: model selection for AR(2) data

```
forecast::auto.arima(ar2)
```

```
Series: ar2
ARIMA(2,0,2) with non-zero mean

Coefficients:
         ar1     ar2     ma1      ma2    mean
      0.2795  0.5938  0.4861  -0.0943  0.9553
s.e.  1.1261  1.0413  1.1284   0.1887  0.3398

sigma^2 estimated as 0.9848:  log likelihood=-1409.57
AIC=2831.15   AICc=2831.23   BIC=2860.59
```

Works with missing data too though might not estimate very close to the true model form.

```
forecast::auto.arima(ar2miss)
```

```
Series: ar2miss
ARIMA(0,1,0)

sigma^2 estimated as 0.5383:  log likelihood=-82.07
AIC=166.15   AICc=166.19   BIC=168.72
```

5.8.2 Fitting to 100 simulated data sets

Let's fit to 100 simulated data sets and see how often the true (generating) model form is selected.

```
save.fits <- rep(NA,100)
for(i in 1:100){
  a2 <- arima.sim(n=100, model=list(ar=c(.8,.1)))
  fit <- auto.arima(a2, seasonal=FALSE, max.d=0, max.q=0)
  save.fits[i] <- paste0(fit$arma[1], "-", fit$arma[2])
}
table(save.fits)
```

```
save.fits
1-0 2-0 3-0
 71  22   7
```

`auto.arima()` uses AICc for selection by default. You can change that to AIC or BIC using `ic="aic"` or `ic="bic"`.

Repeat the simulation using AIC and BIC to see how the choice of the information criteria affects the model that is selected.

5.8.3 Trace=TRUE

We can set `Trace=TRUE` to see what models `auto.arima()` fit.

```
forecast::auto.arima(ar2, trace=TRUE)
```

```
 Fitting models using approximations to speed things up...

 ARIMA(2,0,2) with non-zero mean : 2824.88
 ARIMA(0,0,0) with non-zero mean : 4430.868
 ARIMA(1,0,0) with non-zero mean : 2842.785
 ARIMA(0,0,1) with non-zero mean : 3690.512
 ARIMA(0,0,0) with zero mean     : 4602.31
 ARIMA(1,0,2) with non-zero mean : 2827.422
 ARIMA(2,0,1) with non-zero mean : 2825.235
 ARIMA(3,0,2) with non-zero mean : 2830.176
 ARIMA(2,0,3) with non-zero mean : 2826.503
 ARIMA(1,0,1) with non-zero mean : 2825.438
 ARIMA(1,0,3) with non-zero mean : 2829.358
```

```
 ARIMA(3,0,1) with non-zero mean : 2825.41
 ARIMA(3,0,3) with non-zero mean : 2825.766
 ARIMA(2,0,2) with zero mean     : 2829.536

 Now re-fitting the best model(s) without approximations...

 ARIMA(2,0,2) with non-zero mean : 2831.232

 Best model: ARIMA(2,0,2) with non-zero mean
Series: ar2
ARIMA(2,0,2) with non-zero mean

Coefficients:
         ar1     ar2     ma1      ma2    mean
      0.2795  0.5938  0.4861  -0.0943  0.9553
s.e.  1.1261  1.0413  1.1284   0.1887  0.3398

sigma^2 estimated as 0.9848:  log likelihood=-1409.57
AIC=2831.15   AICc=2831.23   BIC=2860.59
```

5.8.4 stepwise=FALSE

We can set `stepwise=FALSE` to use an exhaustive search. The model may be different than the result from the non-exhaustive search.

```
forecast::auto.arima(ar2, trace=TRUE, stepwise=FALSE)
```

```
 Fitting models using approximations to speed things up...

 ARIMA(0,0,0) with zero mean     : 4602.31
 ARIMA(0,0,0) with non-zero mean : 4430.868
 ARIMA(0,0,1) with zero mean     : 3815.931
 ARIMA(0,0,1) with non-zero mean : 3690.512
 ARIMA(0,0,2) with zero mean     : 3425.037
 ARIMA(0,0,2) with non-zero mean : 3334.754
 ARIMA(0,0,3) with zero mean     : 3239.347
```

```
ARIMA(0,0,3) with non-zero mean : 3170.541
ARIMA(0,0,4) with zero mean     : 3114.265
ARIMA(0,0,4) with non-zero mean : 3059.938
ARIMA(0,0,5) with zero mean     : 3042.136
ARIMA(0,0,5) with non-zero mean : 2998.531
ARIMA(1,0,0) with zero mean     : 2850.655
ARIMA(1,0,0) with non-zero mean : 2842.785
ARIMA(1,0,1) with zero mean     : 2830.652
ARIMA(1,0,1) with non-zero mean : 2825.438
ARIMA(1,0,2) with zero mean     : 2832.668
ARIMA(1,0,2) with non-zero mean : 2827.422
ARIMA(1,0,3) with zero mean     : 2834.675
ARIMA(1,0,3) with non-zero mean : 2829.358
ARIMA(1,0,4) with zero mean     : 2835.539
ARIMA(1,0,4) with non-zero mean : 2829.825
ARIMA(2,0,0) with zero mean     : 2828.987
ARIMA(2,0,0) with non-zero mean : 2823.774
ARIMA(2,0,1) with zero mean     : 2829.952
ARIMA(2,0,1) with non-zero mean : 2825.235
ARIMA(2,0,2) with zero mean     : 2829.536
ARIMA(2,0,2) with non-zero mean : 2824.88
ARIMA(2,0,3) with zero mean     : 2831.461
ARIMA(2,0,3) with non-zero mean : 2826.503
ARIMA(3,0,0) with zero mean     : 2831.057
ARIMA(3,0,0) with non-zero mean : 2826.236
ARIMA(3,0,1) with zero mean     : 2832.662
ARIMA(3,0,1) with non-zero mean : 2825.41
ARIMA(3,0,2) with zero mean     : 2834.788
ARIMA(3,0,2) with non-zero mean : 2830.176
ARIMA(4,0,0) with zero mean     : 2833.323
ARIMA(4,0,0) with non-zero mean : 2828.759
ARIMA(4,0,1) with zero mean     : 2827.798
ARIMA(4,0,1) with non-zero mean : 2823.853
ARIMA(5,0,0) with zero mean     : 2835.315
ARIMA(5,0,0) with non-zero mean : 2830.501

Now re-fitting the best model(s) without approximations...
```

```
 Best model: ARIMA(2,0,0) with non-zero mean

Series: ar2
ARIMA(2,0,0) with non-zero mean

Coefficients:
         ar1     ar2    mean
      0.7684  0.1387  0.9561
s.e.  0.0314  0.0314  0.3332

sigma^2 estimated as 0.9832:  log likelihood=-1409.77
AIC=2827.54   AICc=2827.58   BIC=2847.17
```

5.8.5 Fit to the anchovy data

```
fit <- auto.arima(anchovyts)
fit
```

```
Series: anchovyts
ARIMA(0,1,1) with drift

Coefficients:
          ma1   drift
      -0.6685  0.0542
s.e.   0.1977  0.0142

sigma^2 estimated as 0.04037:  log likelihood=5.39
AIC=-4.79   AICc=-3.65   BIC=-1.13
```

Note `arima()` writes a MA model like:

$$x_t = e_t + b_1 e_{t-1} + b_2 e_{t-2}$$

while many authors use this notation:

$$x_t = e_t - \theta_1 e_{t-1} - \theta_2 e_{t-2}$$

so the MA parameters reported by `auto.arima()` will be NEGATIVE of that reported in Stergiou and Christou (1996) who analyze these same data. Note, in Stergiou and Christou, the model is written in backshift notation on page 112. To see the model as the equation above, I translated from backshift to non-backshift notation.

5.9 Check residuals

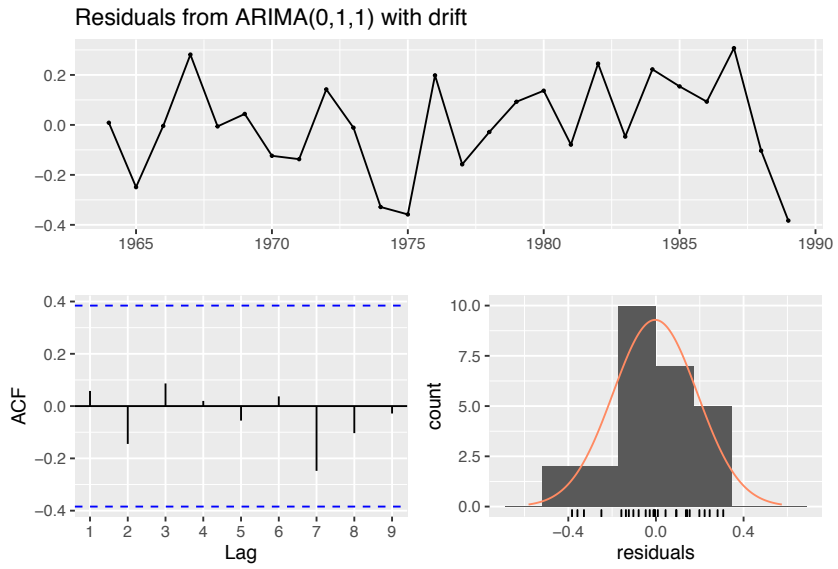
We can do a test of autocorrelation of the residuals with `Box.test()` with `fitdf` adjusted for the number of parameters estimated in the fit. In our case, MA(1) and drift parameters.

```
res <- resid(fit)
Box.test(res, type="Ljung-Box", lag=12, fitdf=2)
```

```
    Box-Ljung test

data:  res
X-squared = 5.1609, df = 10, p-value = 0.8802
```

`checkresiduals()` in the **forecast** package will automate this test and show some standard diagnostics plots.

```
forecast::checkresiduals(fit)
```

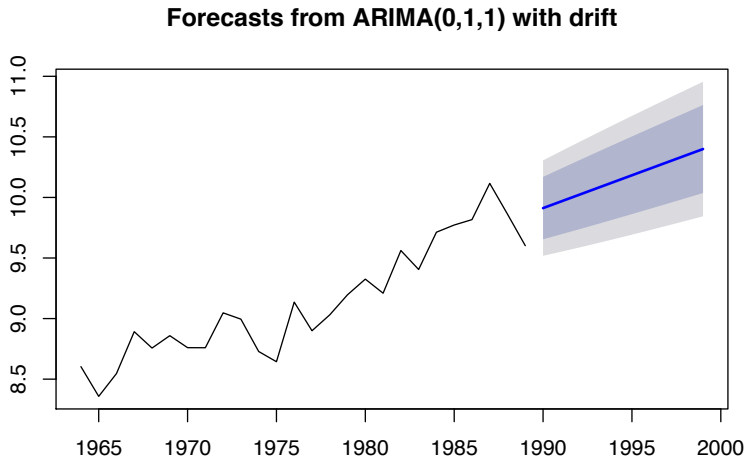



```
    Ljung-Box test

data:  Residuals from ARIMA(0,1,1) with drift
Q* = 1.0902, df = 3, p-value = 0.7794

Model df: 2.   Total lags used: 5
```

5.10 Forecast from a fitted ARIMA model

We can create a forecast from our anchovy ARIMA model using `forecast()`. The shading is the 80% and 95% prediction intervals.

```
fr <- forecast::forecast(fit, h=10)
plot(fr)
```



5.11 Seasonal ARIMA model

The Chinook data are monthly and start in January 1990. To make this into a ts object do

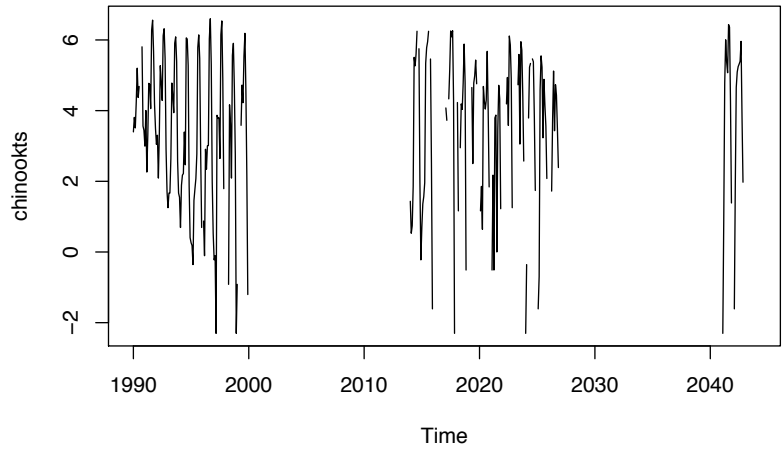
```
chinookts <- ts(chinook$log.metric.tons, start=c(1990,1),
                frequency=12)
```

`start` is the year and month and frequency is the number of months in the year.

Use `?ts` to see more examples of how to set up ts objects.

5.11.1 Plot seasonal data

```
plot(chinookts)
```



5.11.2 `auto.arima()` for seasonal ts

`auto.arima()` will recognize that our data has season and fit a seasonal ARIMA model to our data by default. Let's define the training data up to 1998 and use 1999 as the test data.

```
traindat <- window(chinookts, c(1990,10), c(1998,12))
testdat <- window(chinookts, c(1999,1), c(1999,12))
fit <- forecast::auto.arima(traindat)
fit
```

```
Series: traindat
ARIMA(1,0,0)(0,1,0)[12] with drift

Coefficients:
         ar1    drift
      0.3676  -0.0320
s.e.  0.1335   0.0127

sigma^2 estimated as 0.758:  log likelihood=-107.37
AIC=220.73   AICc=221.02   BIC=228.13
```

Use `?window` to understand how subsetting a ts object works.

5.12 Forecast using a seasonal model

Forecasting works the same using the `forecast()` function.

```
fr <- forecast::forecast(fit, h=12)
plot(fr)
points(testdat)
```