Time Series Analysis

Archit Gupta

September 10, 2016

Contents

Time Series Analysis

The processes that generates observations indexed by time, we refer to these models as time series models. Classic examples of time series are stock market indexes, volume of sales of a company's product over time, and changing weather attributes such as temperature and rainfall during the year.

we will focus on univariate time series, that is to say, time series that involve monitoring how a single variable fluctuates over time. To do this, we begin with some basic tools for describing time series, followed by an overview of a number of fundamental examples.we will focus primarily on ARIMA models.

Fundamental concepts of time series

A time series is just a sequence of random variables, Y1, Y2, ., YT, indexed by an evenly spaced sequence of points in time. Time series are ubiquitous in everyday life; we can observe the total amount of rainfall in millimeters over yearly periods for consecutive years, the average daytime temperature over consecutive days, the price of a particular share in the stock market at the close of every day of trading, or the total number of patients in a doctor's waiting room every half hour.

To analyze time series data, we use the concept of a stochastic process, which is just a sequence of random variables that are generated via an underlying mechanism that is stochastic or random, as opposed to deterministic.

Time series summary functions

- The **mean** function of a time series is the expected value of the time series at a particular time index, t.In the general case, the mean of a time series at a particular time index t1 is not the same as the mean of that time series at a different time index t2.
- The **autocovariance** function and **autocorrelation** function are two important functions that measure the linear dependence between the random variables that make up the time series at different time points. The autocorrelation function is commonly abbreviated to the ACF function. When the two time indexes are the same, the autocovariance function is just the variance.

The variance and autocovariance functions are measured in squared units of the original time series output units. Additionally, the autocovariance is a symmetric function, in that the same result is obtained if we switch the two time indexes in the computation. The ACF function is also symmetric, but it is unitless and its absolute value is bounded by the value 1. When the autocorrelation between two time indexes is 1 or -1, there is a perfect linear dependence or correlation, whereas when the value is 0, then the two time indexes are said to be uncorrelated.

Some fundamental time series

White noise

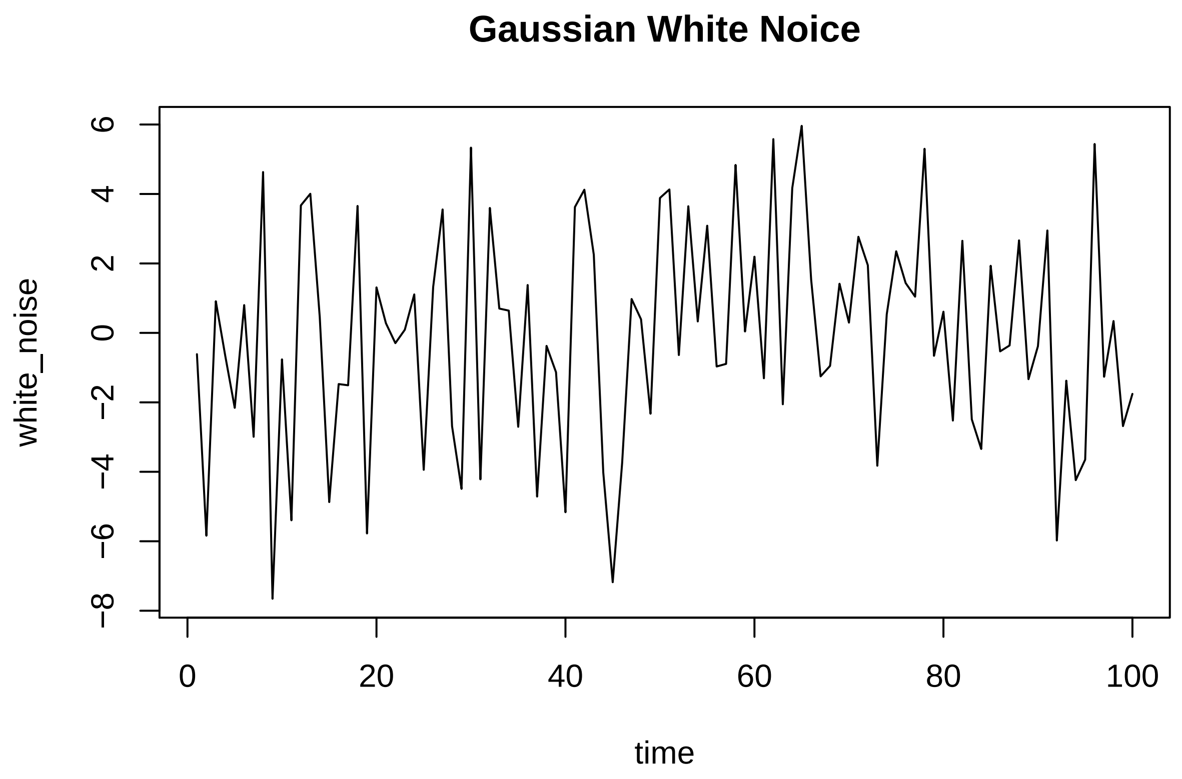
A basic but very important type of time series is known as discrete white noise, or simply white noise. In a white noise time series, the random variables that are generated all have a mean of 0, finite and identical variance ??w2, and the random variables at different time steps are uncorrelated with each other.

a white noise time series drawn from a normal distribution is known as Gaussian white noise. We can easily generate some Gaussian white noise by sampling from a normal distribution with zero mean using the rnorm() function:

```r
set.seed(9571)
white_noise <- rnorm(100, mean = 0, sd = 3.0)
```

The following plot shows our generated Gaussian white noise series:

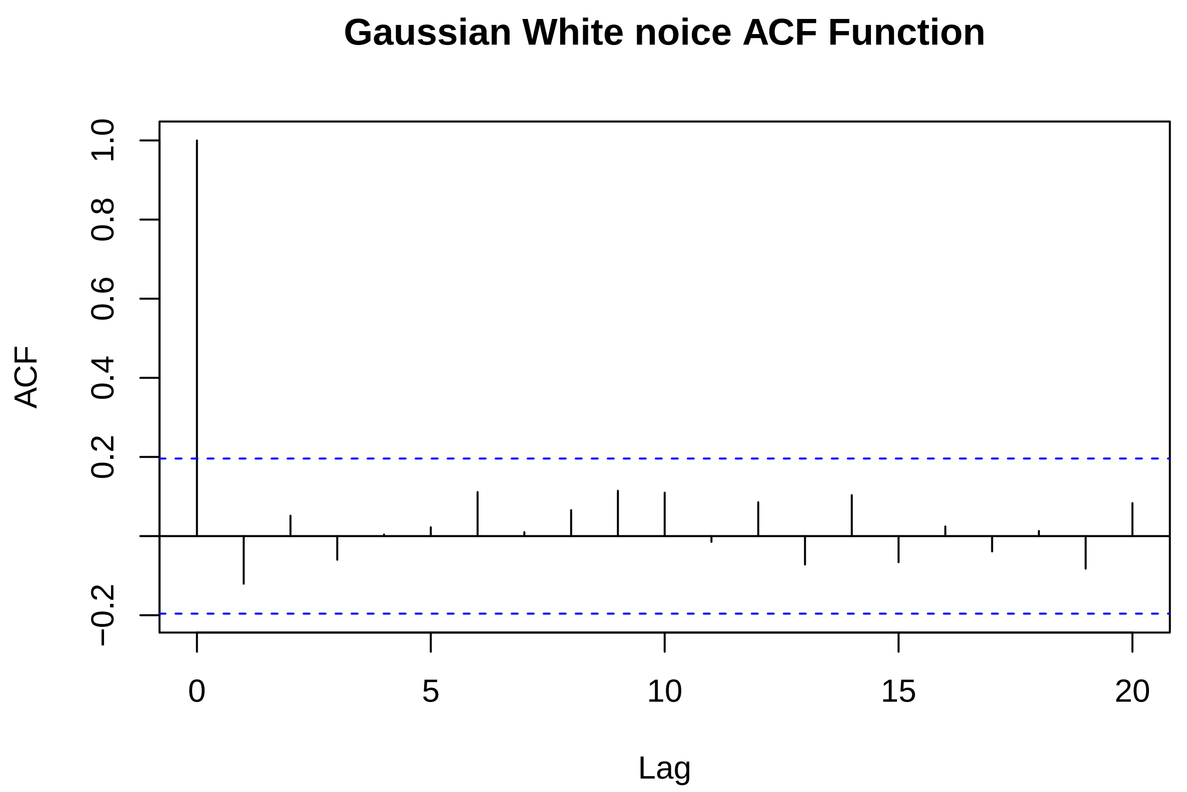
```r
plot(white_noise, xlab="time", main = "Gaussian White Noice", type = "l")
```



Fitting a white noise time series

Suppose we have collected some time series data and want to know if our time series is actually a white noise time series. One way to do this is to plot a correlogram. This is a graph that estimates the ACF function for pairs of time steps that are k steps apart. For different values of k, known as the lag, we should see zero correlation except for the case where k = 0. In this way, we are essentially testing for the uncorrelated property, which is one of the defining characteristics of a white noise series. We can easily do this in R via the acf() function:

```r
acf(white_noise, main = "Gaussian White noice ACF Function")
```



Above is the ACF plot for the white noise series we generated earlier. The dotted lines show a 95 percent confidence interval within which the value of the ACF function at a particular lag is not considered statistically significant (greater than zero). These values in practice will not be exactly zero due to sampling bias and in fact, we can reasonably expect to see 5 percent of them to be erroneously significant for a correlogram of an actual white noise series.

The correlogram we plotted does indeed match our expectations, whereby only the value at a lag of 0 is statistically significant. To fit our observed time series to a white noise series model, all we need is the variance as this is the only parameter of a white noise series. We can estimate this directly via the var() function, which gives a value very close to what we used to generate the sequence:

```
var(white_noise)
```

```
## [1] 9.862699
```

Random walk

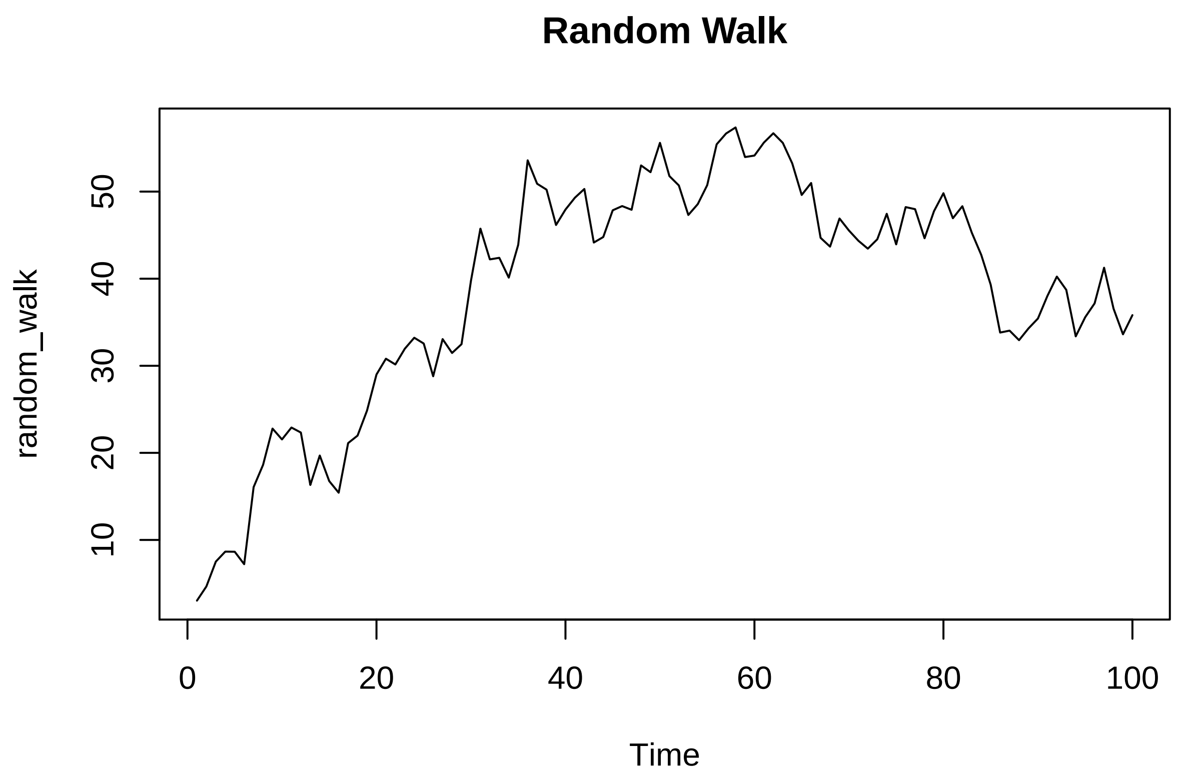
A time series that is a random walk is a time series in which the differences between successive time points are white noise. We can describe this process using a recurrence relation, that is to say, a relation in which a term in the sequence is defined as a function of terms that occur earlier in the sequence.

Essentially, we can think of a random walk as a sequence that adjusts its current value by a positive or negative amount according to a parallel white noise sequence.

To simulate a random walk in R, we essentially need a cumulative sum

```
set.seed(874234242)
random_walk <- cumsum(rnorm(100, mean = 0, sd = 3.0))
```

```r
plot(random_walk, type = "l", xlab = "Time", main = "Random Walk")
```

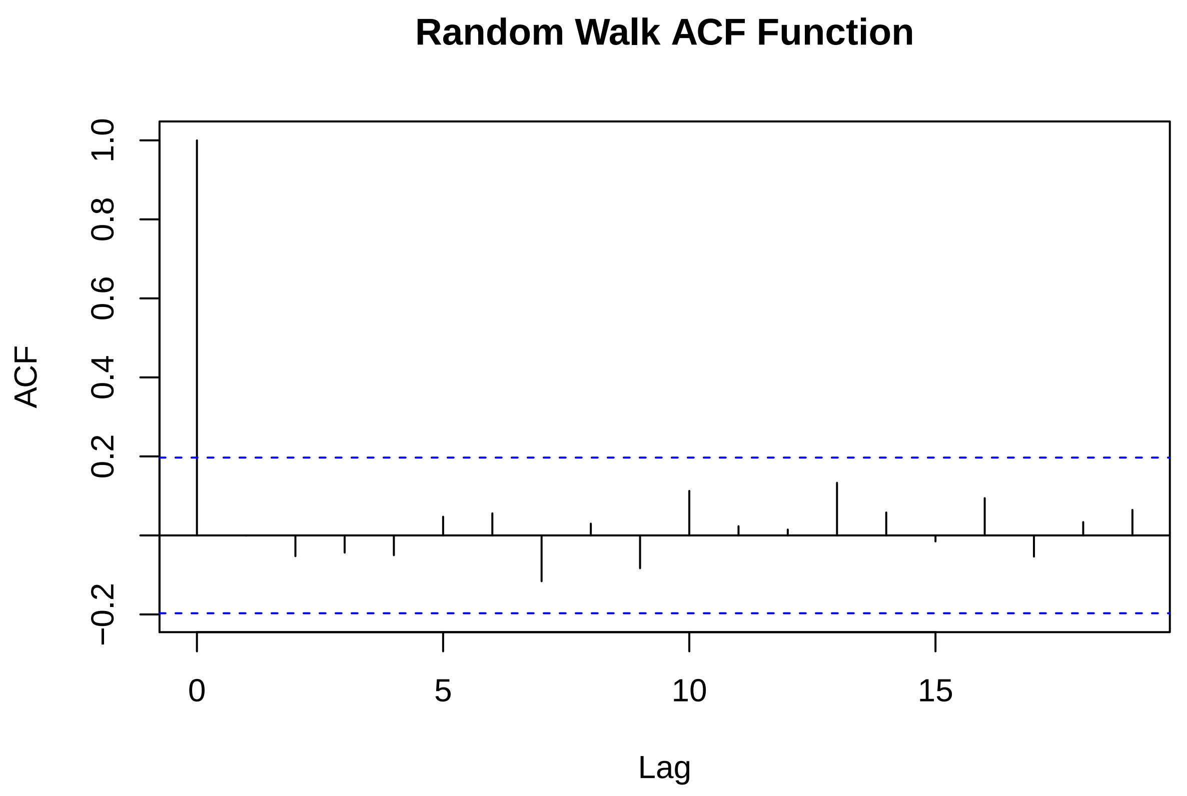


Fitting a random walk

A good way to see if a time series follows a random walk is to compute the successive differences between terms. The time series that is comprised of these differences should be a white noise time series. Consequently, were we to plot a correlogram of the latter, we should see the same kind of pattern characteristic of white noise, namely a spike at lag 0 and no other significant spikes elsewhere.

To compute successive term differences in R, we use the diff() function:

```r
acf(diff(random_walk), main = "Random Walk ACF Function")
```



Stationarity

Stationarity essentially describes that the probabilistic behavior of a time series does not change with the passage of time. There are two versions of the stationarity property that are commonly used. A stochastic process is said to be strictly stationary when the joint probability distribution of a sequence of points starting at time t, Yt, Yt+1, . . . , Yt+n, is the same as the joint probability distribution of another sequence of points starting at a different time T, YT, YT+1, . . . , YT+n.

To be strictly stationary, this property must hold for any choice of time t and T, and for any sequence length n. In particular, because we can choose n = 1, this means that the probability distributions of every individual point in the sequence, also known as the univariate probabilities, must be the same. It follows, therefore, that in a strictly stationary time series, the mean function is constant over time, as is the variance.

White noise is a stationary process as it has a constant mean, in this case of 0, and also a constant variance. Note also that the x axis of the ACF plot displays the time lag k rather than the position in the sequence.

The random walk, on the other hand, does have a constant mean (not so in the case of the random walk with drift however) but it has a time varying variance and so it is non-stationary.

Stationary time series models

Moving average models

A moving average (MA) process is a stochastic process in which the random variable at time step t is a linear combination of the most recent (in time) terms of a white noise process.

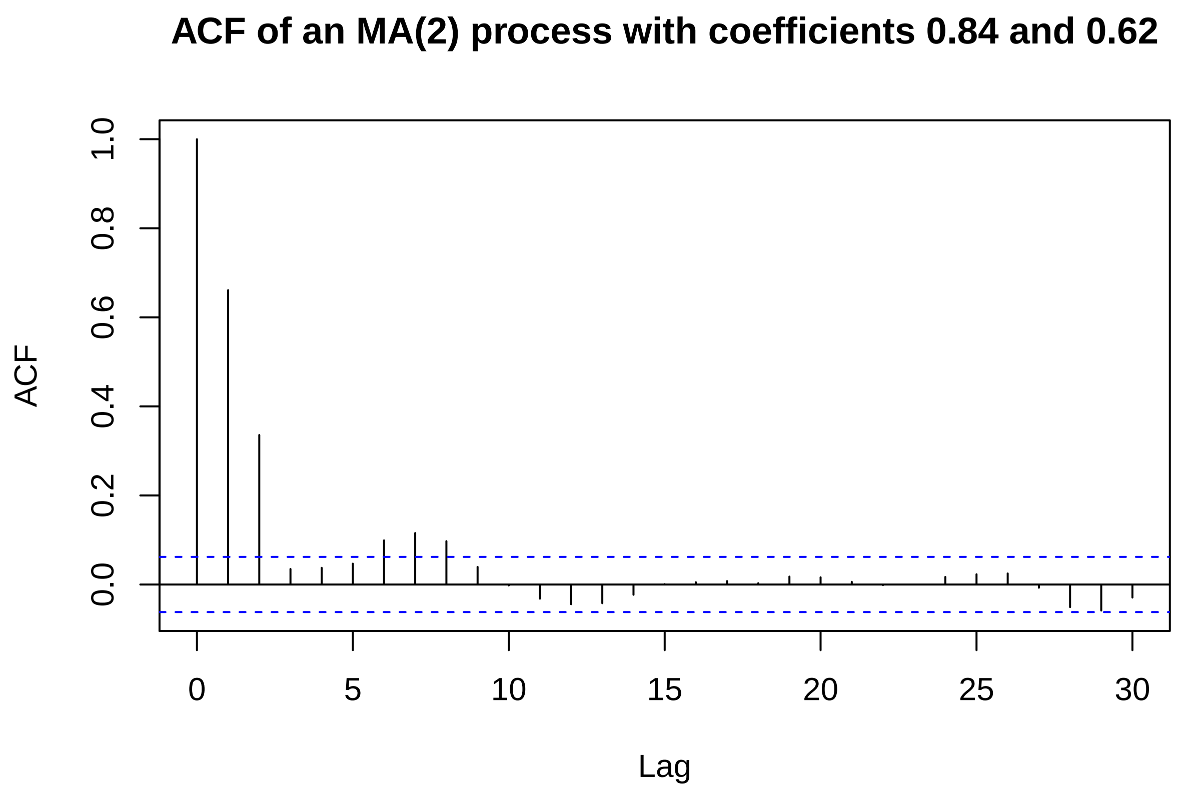
In R, we can simulate MA processes (among others) using the arima.sim() function. To generate an MA process, we will use the n parameter to specify the length of the series and the model parameter to specify the parameters of the series we want to simulate. This, in turn, is a list in which we will set the vector of ?? coefficients in the ma attribute and the standard deviation of the white noise terms in the sd attribute.

```
set.seed(2357977)
ma_ts1 <- arima.sim(model = list(ma = c(0.84, 0.62), sd = 1.2), n = 1000)
head(ma_ts1, n = 8)
```

```
## [1] -2.403431159 -2.751889402 -2.174711499 -1.354482419 -0.814139443  0.009842499 -0.632004838 -0.03
```

The arima.sim() function returns the result of the simulation in a special time series object that R calls ts. This object is useful for keeping track of some basic information about a time series and supports specialized plots that are useful in time series analysis. According to what we learned in this section, the ACF plot of this simulated time series should display two significant peaks at lags 1 and 2.

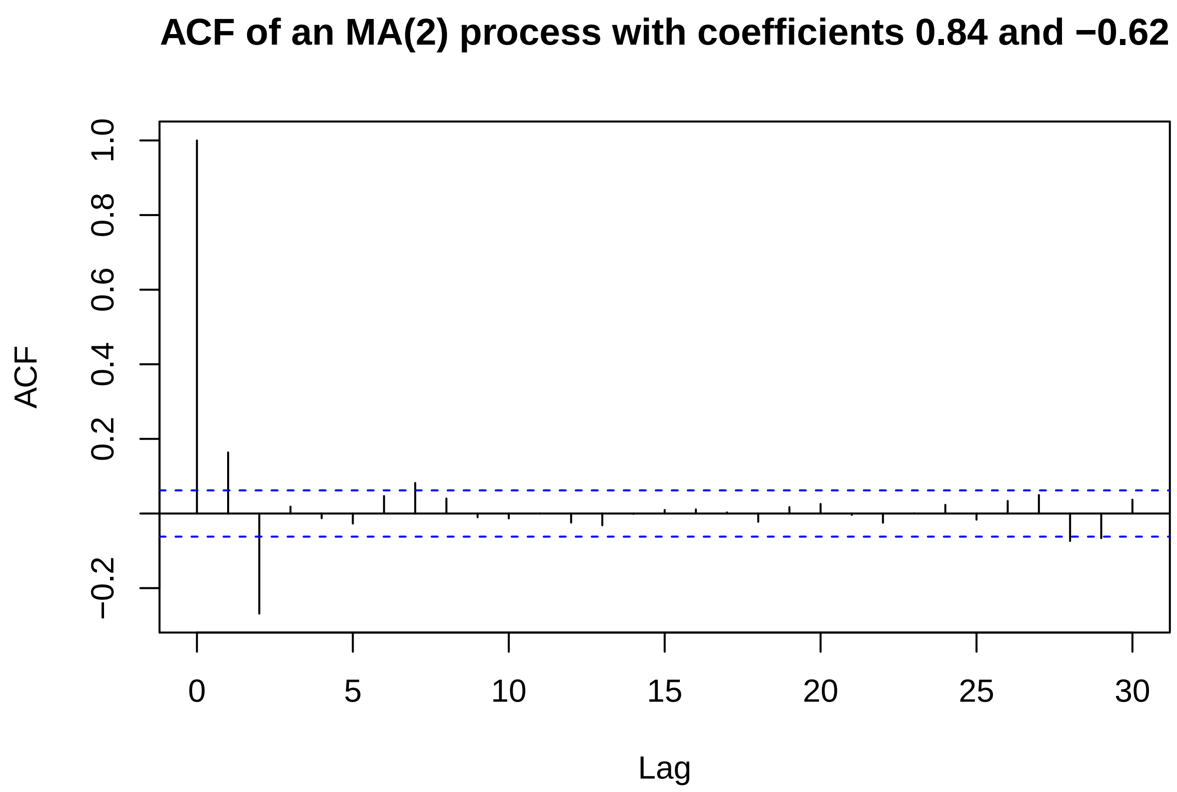
```
acf(ma_ts1, main = "ACF of an MA(2) process with coefficients 0.84 and 0.62")
```



```
set.seed(2357977)
ma_ts1 <- arima.sim(model = list(ma = c(0.84, -0.62), sd = 1.2), n = 1000)
head(ma_ts1, n = 8)
```

```
## [1] -2.5324158 -2.2548188  0.4679743 -0.4701794 -0.4987769  0.8762283 -0.5457608 -0.6574361
```

```
acf(ma_ts1, main = "ACF of an MA(2) process with coefficients 0.84 and -0.62")
```



Autoregressive models

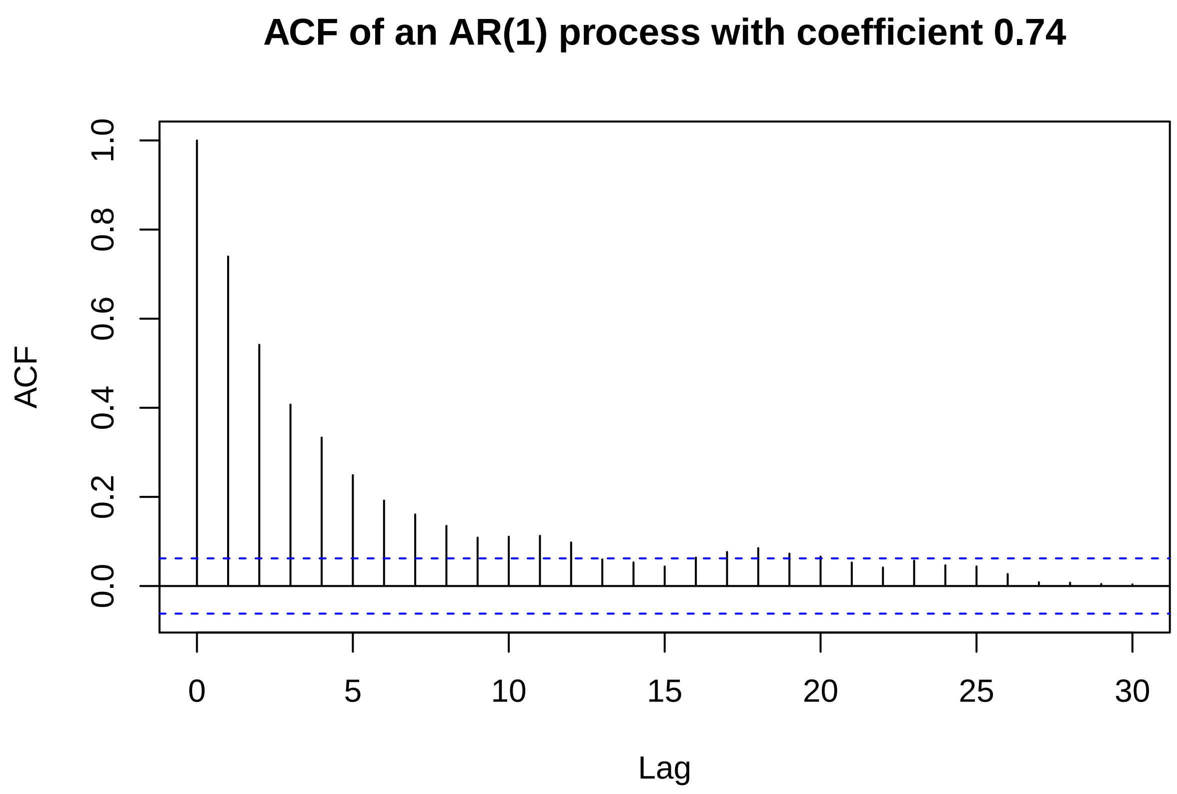
Autoregressive models (AR) come about from the notion that we would like a simple model to explain the current value of a time series in terms of a limited window of the most recent values from the past.

We will simulate an AR(1) process with the arima.sim() function again. This time, the coefficients of our process are defined in the ar attribute inside the model list.

```
set.seed(634090)
ma_ts3 <- arima.sim(model = list(ar = c(0.74), sd = 1.2),n = 1000)
```

The following ACF plot shows the exponential decay in lag coefficients, which is what we expect for our simulated AR(1) process:

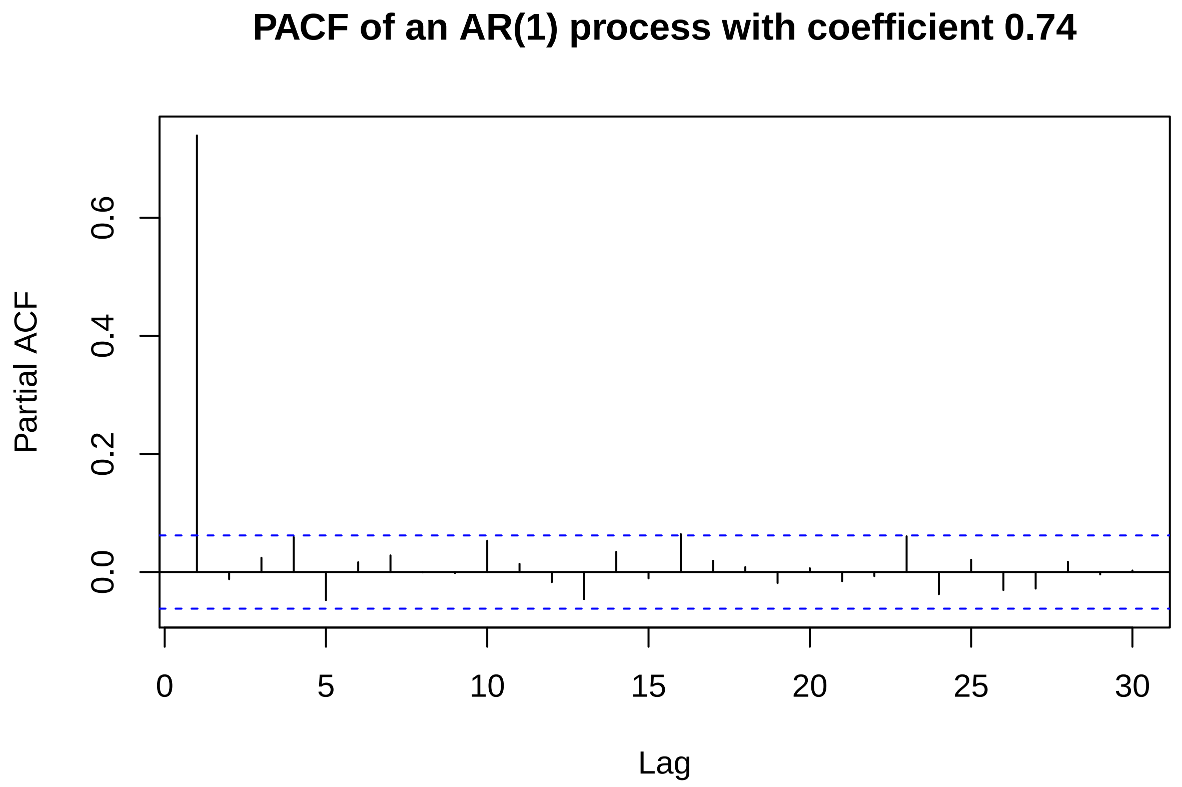
```
acf(ma_ts3, main = "ACF of an AR(1) process with coefficient 0.74")
```



Note that with the moving average process, the ACF plot was useful in helping us identify the order of the process. With an autoregressive process, this is not the case. To deal with this, we use the partial autocorrelation function (PACF) plot instead. We define the partial autocorrelation at time lag, k, as the correlation that results when we remove the effect of any correlations that are present for lags smaller than k. By definition, the AR process of order p depends only on the values of the process up to p units of time in the past.

Consequently, the PACF plot will exhibit zero values for all the lags greater than p, creating a parallel situation with the ACF plot for an MA process where we can simply read off the order of an AR process as the largest time lag whose PACF lag term is statistically significant. In R, we can generate the PACF plot with the function pacf(). For our AR(1) process, this produces a plot with only one significant lag term as expected:

```
pacf(ma_ts3, main = "PACF of an AR(1) process with coefficient 0.74")
```



# Non-stationary time series models

Autoregressive integrated moving average models

we can say that a dth order difference is obtained by repeatedly computing differences between consecutive terms d times, to obtain a new sequence with points, Wt, from an original sequence, Yt.

autoregressive integrated moving average (ARIMA) process is a process in which the d order difference of terms is a stationary ARMA process. An ARIMA(p, d, q) process requires a dth order differencing, has an MA component of order q, and an AR component of order p. Thus, a regular ARMA(p, q) process is equivalent to an ARIMA(p, 0, q) process. In order to fit an ARIMA model, we need to first determine an appropriate value of d, namely the number of times we need to perform differencing. Once we have found this, we can then proceed with fitting the differenced sequence using the same process as we would use with an ARMA process.

One way to find a suitable value for d is to repeatedly difference a time series and after each application of differencing to check whether the resulting time series is stationary. There are a number of tests for stationarity that can be used, and these are often known as unit root tests. A good example is the Augmented Dickey-Fuller (ADF) test

We can find an implementation of this test in the R package tseries via the function adf.test(). This assumes a default value k equal to the largest integer that does not exceed the cube root of the length of the time series under test. The ADF test produces a p-value for us to examine. Values that are smaller than 0.05 (or a smaller cutoff such as 0.01 if we want a higher degree of confidence) are typically suggestive that the time series in question is stationary.

The following example shows the results of running the ADF test on our simulated random walk, which we know is non-stationary:

```
library(tseries)
adf.test(random_walk, alternative = "stationary")
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  random_walk
## Dickey-Fuller = -1.5881, Lag order = 4, p-value = 0.7473
## alternative hypothesis: stationary
```

As expected, our p-value is much higher than 0.05, indicating a lack of stationarity.

Casestudy : Predicting intense earthquakes

Our data set is a time series of earthquakes having magnitude that exceeds 4.0 on the Richter scale in Greece over the period between the year 2000 and the year 2008. This data set was recorded by the Observatory of Athens and is hosted on the website of the University of Athens, Faculty of Geology, Department of Geophysics & Geothermics. The data is available online at *http://www.geophysics.geol.uoa.gr/catalog/catgr_20002008.epi*

We will import these data directly by using the package RCurl. From this package, we will use the functions getURL(), which retrieves the contents of a particular address on the Internet, and textConnection(), which will interpret the result as raw text. Once we have the data, we provide meaningful names for the columns using information from the website:

```
library("RCurl")
```

```
## Loading required package: bitops
```

```
seismic_raw <- read.table(textConnection(getURL("http://www.geophysics.geol.uoa.gr/catalog/catgr_200020

names(seismic_raw) <- c("date", "mo", "day", "hr", "mn", "sec","lat", "long","depth", "mw")

head(seismic_raw, n = 3)
```

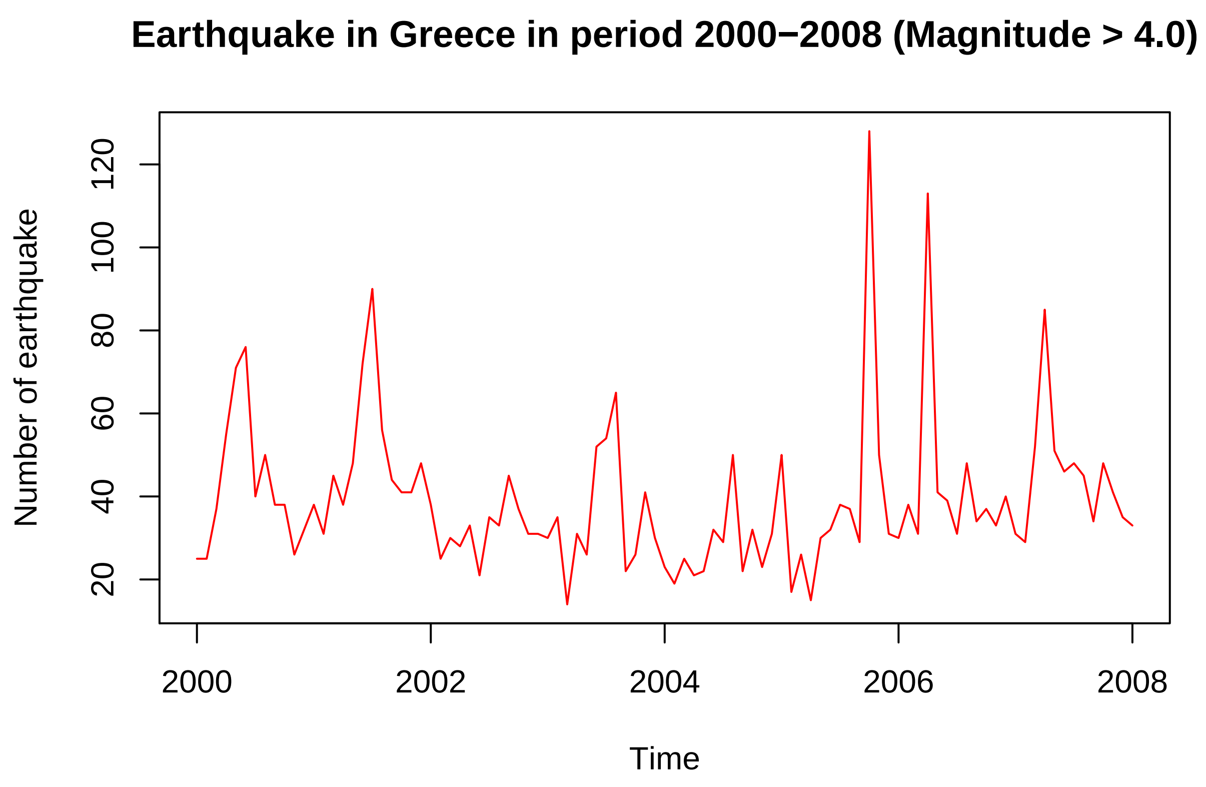
```
##   date mo day hr mn  sec     lat  long depth  mw
## 1 2000  1   1  1 19 28.3 41.950N 20.63     5 4.8
## 2 2000  1   1  4  2 28.4 35.540N 22.76    22 3.7
## 3 2000  1   2 10 44 10.9 35.850N 27.61     3 3.7
```

The first column is the date column and the second is the month column. The next four columns represent the day, hour, minute, and second that the earthquake was observed. The next two columns are the latitude and longitude of the epicenter of the earthquake. The last two columns contain the surface depth and earthquake intensity.

Our goal is to aggregate these data in order to get monthly counts of the significant earthquakes observed in this time period. To achieve this, we can use the count() function of the R package plyr to aggregate the data, and the standard ts() function to create a new time series. We will specify the starting and ending year and month of our time series, as well as set the freq parameter to 12 to indicate monthly readings.

```
library("plyr")
seismic <- count(seismic_raw, c("date", "mo"))
seismic_ts <- ts(seismic$freq
                 , start = c(2000, 1)
                 ,end = c(2008, 1)
                 , frequency = 12)
```

```r
plot(seismic_ts
     , type = "l"
     , ylab = "Number of earthquake"
     , main = "Earthquake in Greece in period 2000-2008 (Magnitude > 4.0)"
     , col = "red")
```



The data seems to fluctuate around 30 with occasional peaks, and although the largest two of these are more recent in time, there does not appear to be any overall upward trend.

We would like to analyze these data using an ARIMA model; however, at the same time, we are not sure what values we should use for the order. A simple way that we can compute this ourselves is to obtain the AIC for all the different models we would like to train and pick the model that has the smallest AIC. Concretely, we will first begin by creating ranges of possible values for the order parameters p, d, and q. Next, we shall use the expand.grid() function, which is a very useful function that will create a data frame with all the possible combination of these parameters:

```r
d <- 0 : 2
p <- 0 : 6
q <- 0 : 6
seismic_models <- expand.grid(d = d, p = p, q = q)
head(seismic_models, n = 4)
```

```
##   d p q
## 1 0 0 0
## 2 1 0 0
## 3 2 0 0
## 4 0 1 0
```

Next, we define a function that fits an ARIMA model using a particular combination of order parameters and returns the AIC produced:

```
getTSModelAIC <- function(ts_data, p, d, q) {
  ts_model <- arima(ts_data, order = c(p, d, q))
  return(ts_model$aic)
}
```

For certain combinations of order parameters, our function will produce an error if it fails to converge. When this happens, we'll want to report a value of infinity for the AIC value so that this model is not chosen when we try to pick the best model. The following function acts as a wrapper around our previous function:

```
getTSModelAICSafe <- function(ts_data, p, d, q) {
  result = tryCatch({
    getTSModelAIC(ts_data, p, d, q)
    }, error = function(e) {
        Inf
  })
}
```

All that remains is to apply this function on every parameter combination in our seismic_models data frame, save the results, and pick the combination that gives us the lowest AIC:

```
seismic_models$aic <- mapply(function(x, y, z)
  getTSModelAICSafe(seismic_ts, x, y, z)
  , seismic_models$p
  ,seismic_models$d
  , seismic_models$q)
```

```
## Warning in log(s2): NaNs produced

## Warning in arima(ts_data, order = c(p, d, q)): possible convergence problem: optim gave code = 1

## Warning in log(s2): NaNs produced

## Warning in arima(ts_data, order = c(p, d, q)): possible convergence problem: optim gave code = 1

## Warning in arima(ts_data, order = c(p, d, q)): possible convergence problem: optim gave code = 1

## Warning in arima(ts_data, order = c(p, d, q)): possible convergence problem: optim gave code = 1
```

```
subset(seismic_models,aic == min(aic))
```

```
##    d p q     aic
## 26 1 1 1 832.171
```

The results indicate that the most appropriate model for our earthquakes time series is the ARIMA(1, 1, 1) model. We can train this model again with these parameters:

```
seismic_model <- arima(seismic_ts, order = c(1, 1, 1))
summary(seismic_model)
```

```
##           Length Class  Mode
## coef       2     -none- numeric
## sigma2     1     -none- numeric
## var.coef   4     -none- numeric
## mask       2     -none- logical
## loglik     1     -none- numeric
## aic        1     -none- numeric
## arma       7     -none- numeric
## residuals 97     ts     numeric
## call       3     -none- call
## series     1     -none- character
## code       1     -none- numeric
## n.cond     1     -none- numeric
## nobs       1     -none- numeric
## model     10     -none- list
```

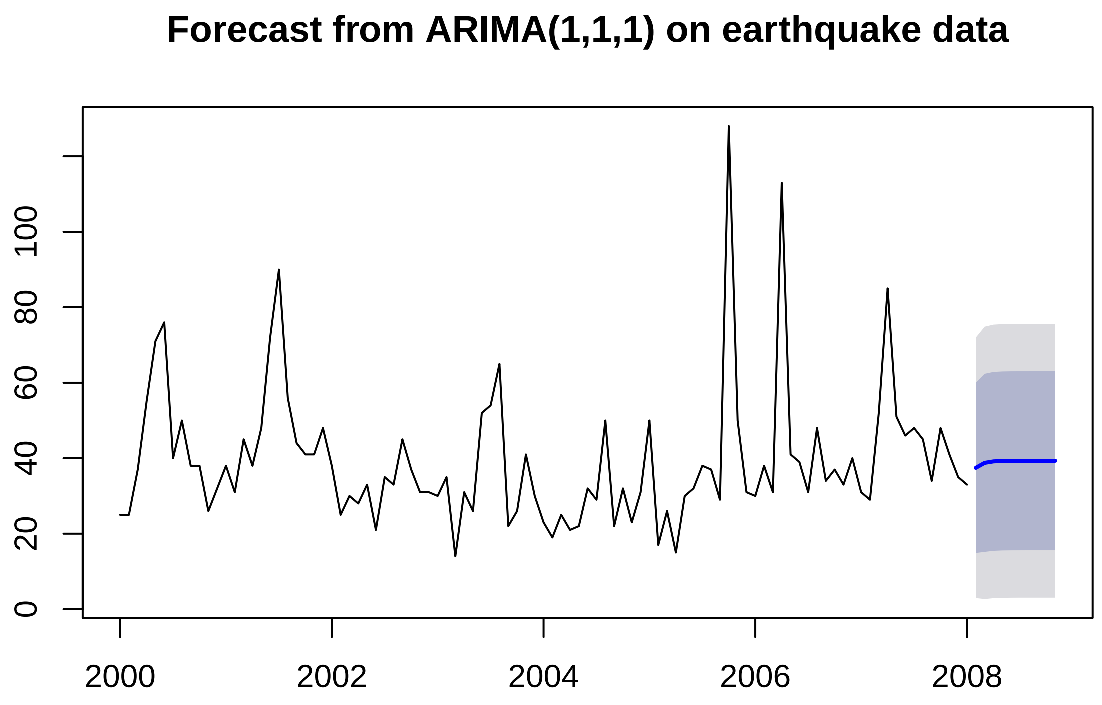
The forecast package has a very useful forecasting function, forecast, that can be applied to time series models. This, in turn, provides us with not only a convenient method to forecast values into the future, but also allows us to visualize the result using the plot() function.

Let's forecast the next ten points in the future:

```r
library(forecast)
```

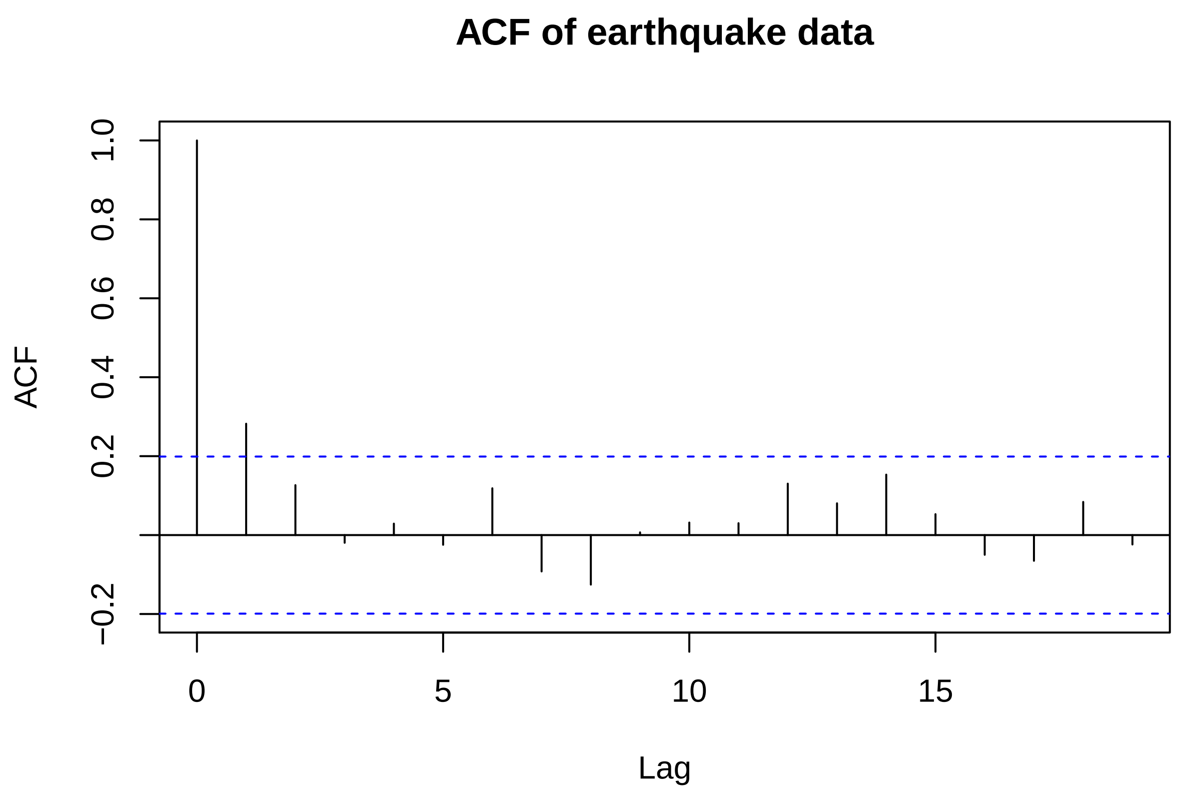
```
## Loading required package: zoo

##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

## Loading required package: timeDate

## This is forecast 7.2
```

```r
plot(forecast(seismic_model, 10)
     , main = "Forecast from ARIMA(1,1,1) on earthquake data")
```



The plot shows that our model predicts a small rise in the number of intense earthquakes over the next few time periods and then predicts a steady average value. Note that the bands shown are confidence intervals around these forecasts. The spiky nature of the time series is reflected in the width of these bands. If we examine the autocorrelation function of our time series, we see that there is only one lag coefficient that is statistically significant:

```r
acf(seismic[3], main = "ACF of earthquake data")
```



The profile of the ACF indicates that a simple AR(1) process might be an appropriate fit, and indeed this is what is predicted using the auto.arima() function of the forecast package.As a final note, we will add that if we want to inspect the coefficients of our fitted model, we can do so using the coefficients() function.

```
coefficients(seismic_model)
```

```
##        ar1        ma1
##  0.2948803 -0.9999997
```

Summary

We started off by looking at some properties of time series such as the autocorrelation function and saw how this, along with the partial autocorrelation function, can provide important clues about the underlying process involved.

Next, we introduced stationarity, which is a very useful property of some time series that in a nutshell says that the statistical behavior of the underlying process does not change over time. We introduced white noise as a stochastic process that forms the basis of many other processes. In particular, it appears in the random walk process, the moving average (MA) process, as well as the autoregressive process (AR). These, in turn, we saw can be combined to yield even more complex time series.

In order to handle the non-stationary case, we introduced the ARIMA process, which tries to render a time series stationary through differencing.