Tutorial: Simulating and Estimating ARMA models

by Kevin Kotzé

1 Basic setup for most empirical work

To open the project for this tutorial, extract the files from the zip folder `T2-arma.zip` and open the `T2-arma.Rproj` file. The first program for this session, is called `T2_arma.R`. After providing a brief description of what this program seeks to achieve, the first thing that we usually do is clear all variables from the current environment and close all the plots. This is performed with the following commands:

```
rm(list=ls())
graphics.off()
```

Thereafter, we will install the additional that we need for this session. There are a few routines that I've compiled in a package that is contained on my **GitHub** account. To install this package, which is named `tsm` you need to run the following commands:

```
devtools::install_github("KevinKotze/tsm")
devtools::install_gitlab("KevinKotze/sarb2020q1")
install.packages("fArma")
install.packages("forecast")
```

Note that at various points in time, some packages that have been hosted on **CRAN** may fail the maintainers unit testing. When this happens, the `install.packages()` command will give you an error message. The simplest solution to this problem is usually to install the package from the **CRAN** GitHub repository, so you would make use of the command `devtools::install_github("cran/fArma")` if you received an error when trying to install the `fArma` package. If this does not work then you need to search on the web for the original repository for this package so that you can install it from that location.

The next step is to make sure that you can access the routines in this package by making use of the `library` command.

```
library(tidyverse)
library(tsm)
```

2 Simulated processes

Now we are good to go! Let's generate 200 observations for a number of different simulated autoregressive, moving average and ARMA time series processes. To ensure that we all get the same results, we set the seed to a predetermined value before we generate values for the respective variables. All of these variables will be stored within a `tibble()` object.

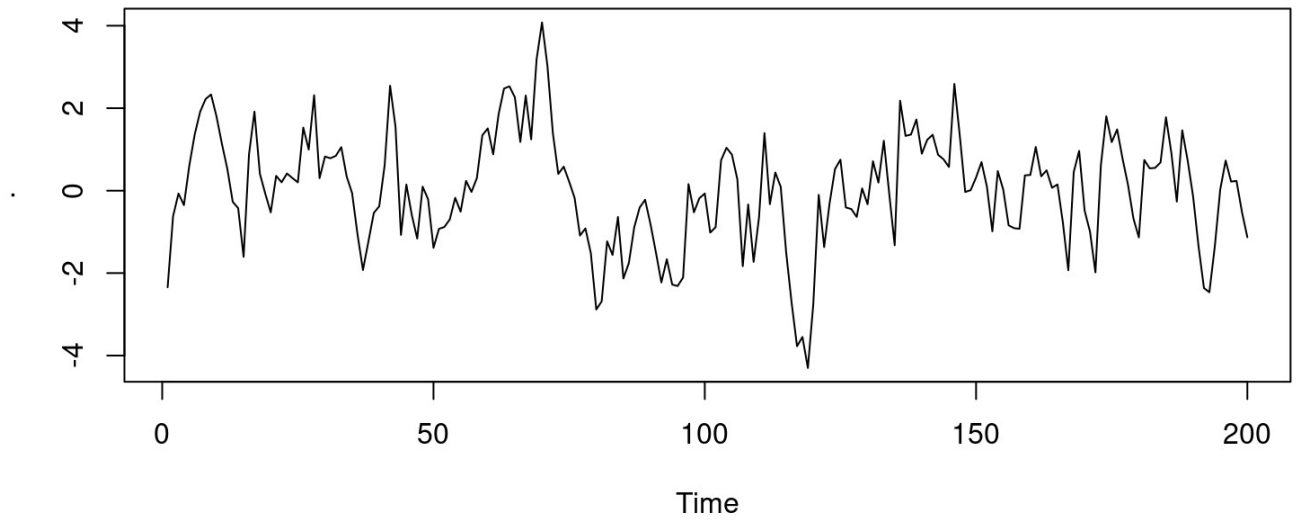
```
set.seed(123)

dat <- tibble(
  ar1 = arima.sim(model = list(ar = 0.8), n = 200),
  ma1 = arima.sim(model = list(ma = 0.7), n = 200),
  arma10 = arima.sim(model = list(ar  = 0.4), n = 200),
  arma01 = arima.sim(model = list(ma = 0.5), n = 200),
  arma11 = arima.sim(model = list(ar = 0.4, ma = 0.5), n = 200),
  arma21 = arima.sim(model = list(ar = c(0.6,-0.2), ma = c(0.4)), n = 200)
)
```

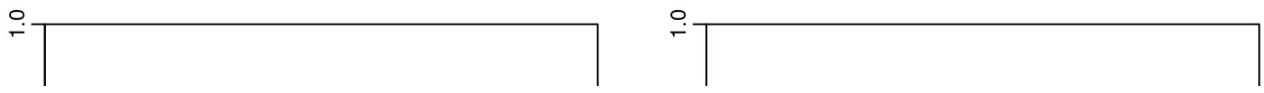
3 Simulated autoregressive process

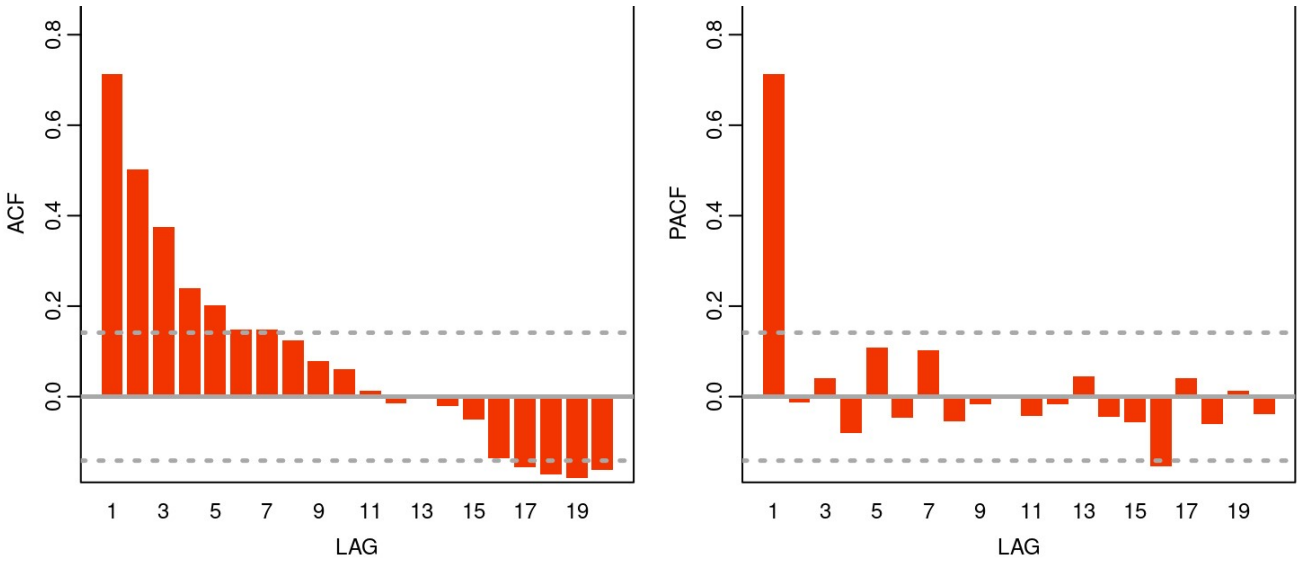
To plot the autoregressive process we can use the `plot.ts`, which is quick and easy to use. Thereafter, we can take a look at autocorrelation and partial autocorrelation function for this variable using the `ac` command. In both cases, we are going to `pull()` the data that we want to use out of the `tibble()` object.

```
dat %>%
  pull(ar1) %>%
  plot.ts()
```




```
dat %>%
  pull(ar1) %>%
  ac(., max.lag = 20)
```





If you want to make use of a more attractive graph then you can take a look at the `ggplot2` package, which is a part of the tidyverse.

To display the Ljung-Box statistic for the first lag we would execute the command, where we note that the *p*-value suggests that the autocorrelation is different from zero.

```
dat %>%
  pull(ar1) %>%
  Box.test(., lag = 1, type = "Ljung-Box")
```

```
##
##  Box-Ljung test
##
## data:  .
## X-squared = 103.29, df = 1, p-value < 2.2e-16
```

Similarly, we could assign the output from the test to an object `Box_res` .

```
Box_res <- dat %>%
  pull(ar1) %>%
  Box.test(., lag = 1, type = "Ljung-Box")
```

To get the result from the object we could just type:

```
Box_res
```

```
##
##  Box-Ljung test
##
## data:  .
## X-squared = 103.29, df = 1, p-value < 2.2e-16
```

Or alternatively you could use `print(Box_res)`. To check what is in the object `Box_res` we could make use of the `names()` command:

```
names(Box_res)
```

```
## [1] "statistic" "parameter" "p.value"   "method"
## [5] "data.name"
```

If we then want to create a vector of *Q*-statistics for the first 10 lags we can assign values to a object we would create a vector for the `Q_stat` and `Q_prob` values.

```
Q_stat <- rep(0, 10)
Q_prob <- rep(0, 10)
```

Thereafter, we assign values for the Q-stat to element places in the different objects. For the first statistic:

```
Q_stat[1] <- Box.test(dat$ar1, lag=1, type="Ljung-Box")$statistic
Q_prob[1] <- Box.test(dat$ar1, lag=1, type="Ljung-Box")$p.value
```

And to view the data in the respective vectors we just type:

```
Q_stat
```

```
##  [1] 103.2929   0.0000   0.0000   0.0000   0.0000
##  [6]   0.0000   0.0000   0.0000   0.0000   0.0000
```

```
Q_prob
```

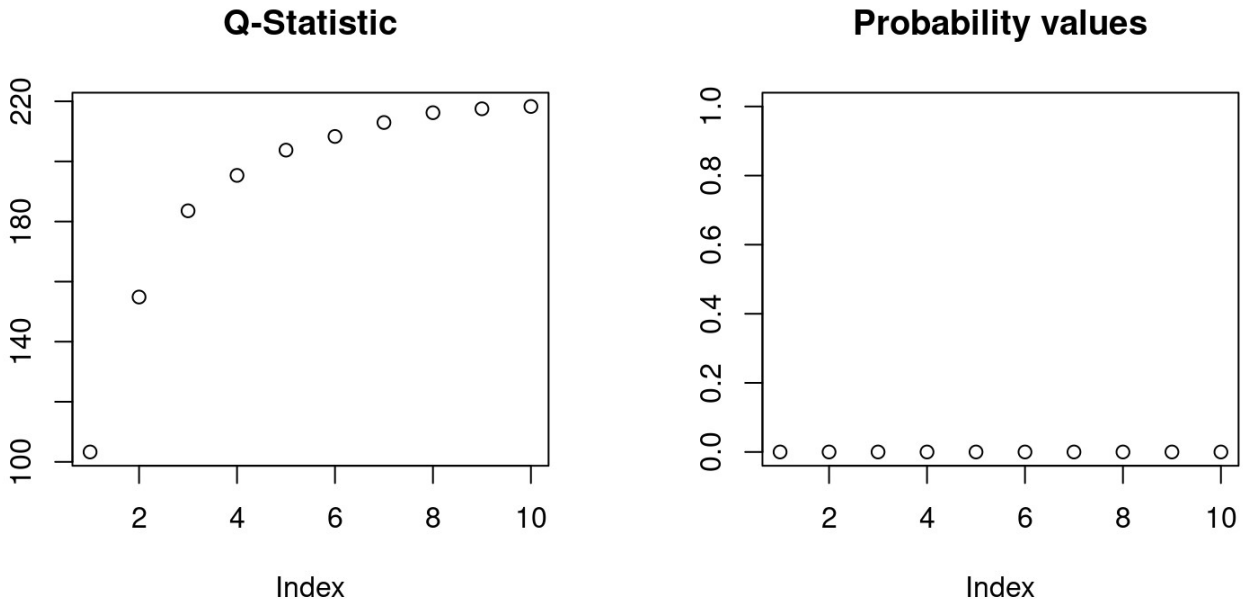
```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Of course, rather than writing code to assign each value individually we can use the loop that takes the form:

```
for (i in 1:10) {
  Q_stat[i] <- Box.test(dat$ar1, lag=i, type="Ljung-Box")$statistic
  Q_prob[i] <- Box.test(dat$ar1, lag=i, type="Ljung-Box")$p.value
}
```

To graph both of these statistics together we could make use of the code:

```
op <- par(mfrow = c(1, 2)) # create plot area of (1 X 2)
 plot(Q_stat, ylab = "", main = 'Q-Statistic')
 plot(Q_prob, ylab = "", ylim = c(0,1), main = 'Probability values')
```




```
par(op) # close plot area
```

If you made use of `ggplot` figures then the `patchwork` package provides a convenient way to combine figures.

These *p*-value values suggest that there is significant autocorrelation in this time series process. To speed up the above computation you should vectorise your code and create a function before using the `map` function in the `purrr` package.

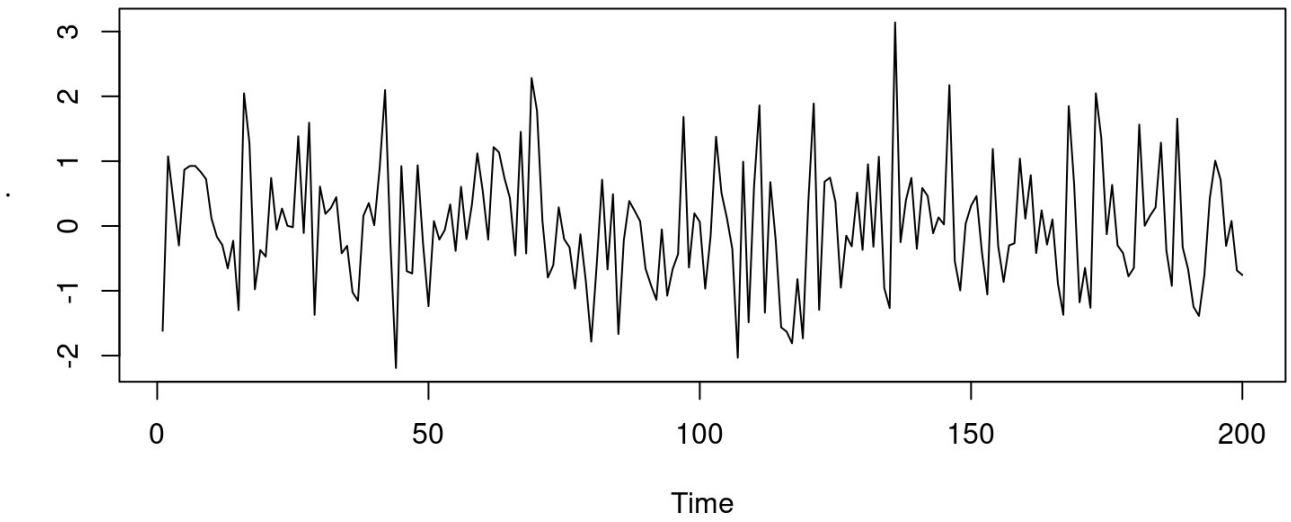
We can now fit a model to this process with the aid of an AR(1) specification and look at the results that are stored in the `arma10` object that we have created.

```
arma10 <- dat %>%
  pull(ar1) %>%
  arima(., order = c(1,0,0), include.mean = FALSE) # uses ARIMA(p,d,q) specification
arma10
```

```
##
## Call:
## arima(x = ., order = c(1, 0, 0), include.mean = FALSE)
##
## Coefficients:
##          ar1
##       0.7234
## s.e.  0.0491
##
## sigma^2 estimated as 0.8744:  log likelihood = -270.74,  aic = 545.48
```

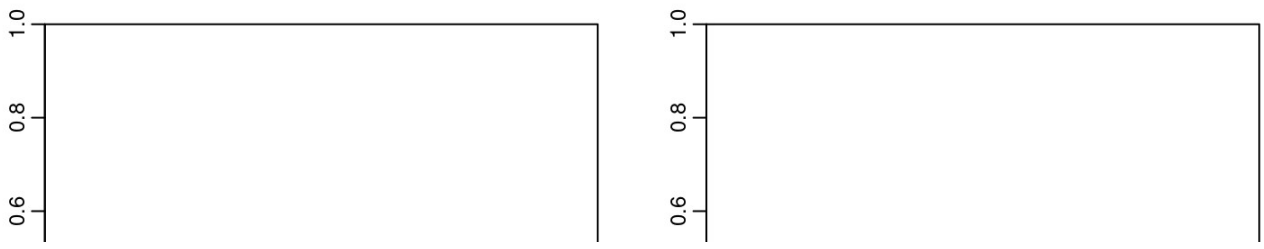
To take a look at what the residuals look like we plot the residuals that are stored in the `arma10` object.

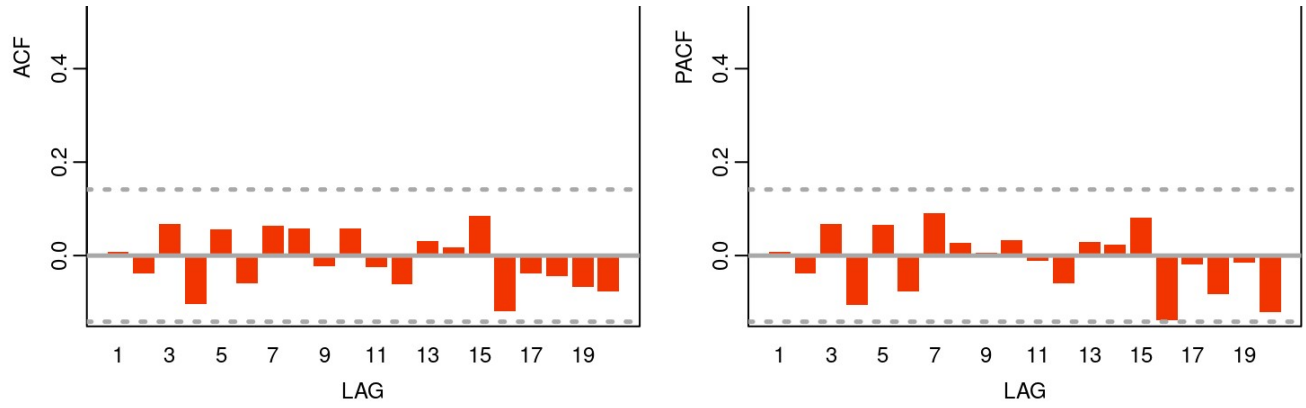
```
par(mfrow=c(1, 1))
arma10$residuals %>%
  plot()
```



Now lets take a look at the ACF and PACF for the residuals

```
arma10$residuals %>%
  ac(., max.lag=20)
```



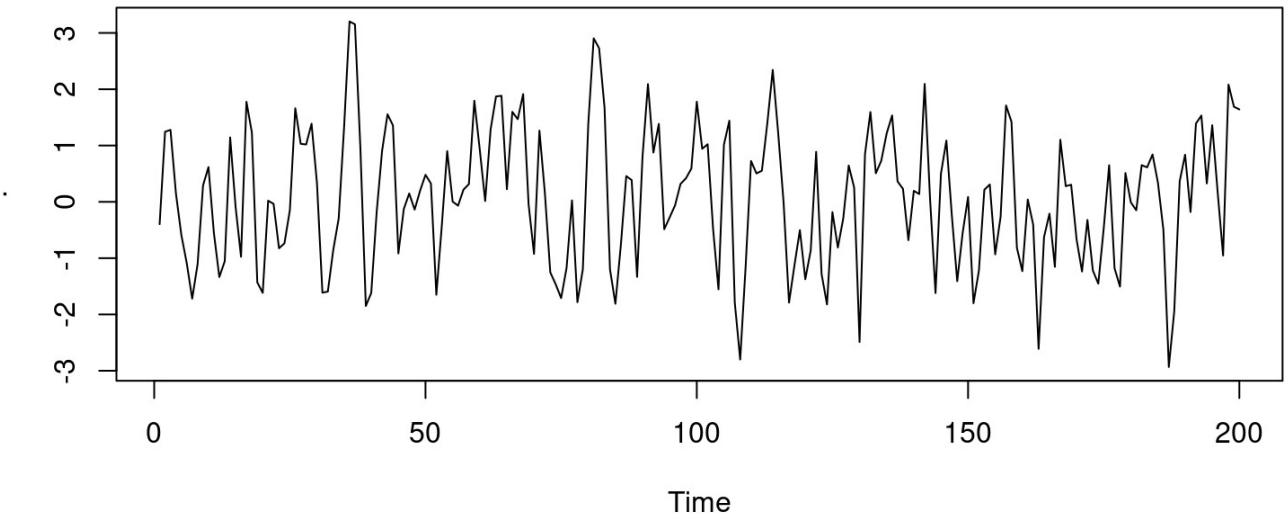


There would appear to be no serial correlation in the residual, which we could confirm with *Q*-statistics, where we should impose a correction for the degrees of freedom.

4 Simulated moving average process

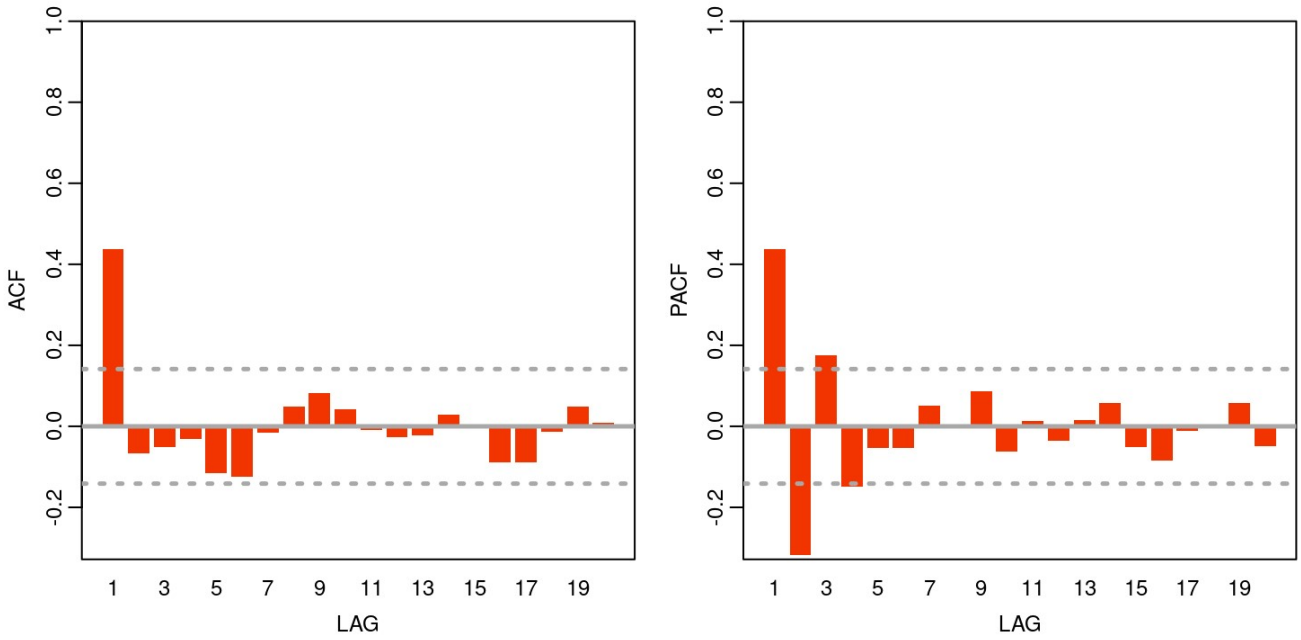
As is always the case, the first thing that we do is visually inspect the data.

```
par(mfrow=c(1, 1))
dat %>%
  pull(ma1) %>%
  plot.ts()
```



We can then consider the ACF and PACF for this variable.

```
dat %>%
  pull(ma1) %>%
  ac(., max.lag=20)
```



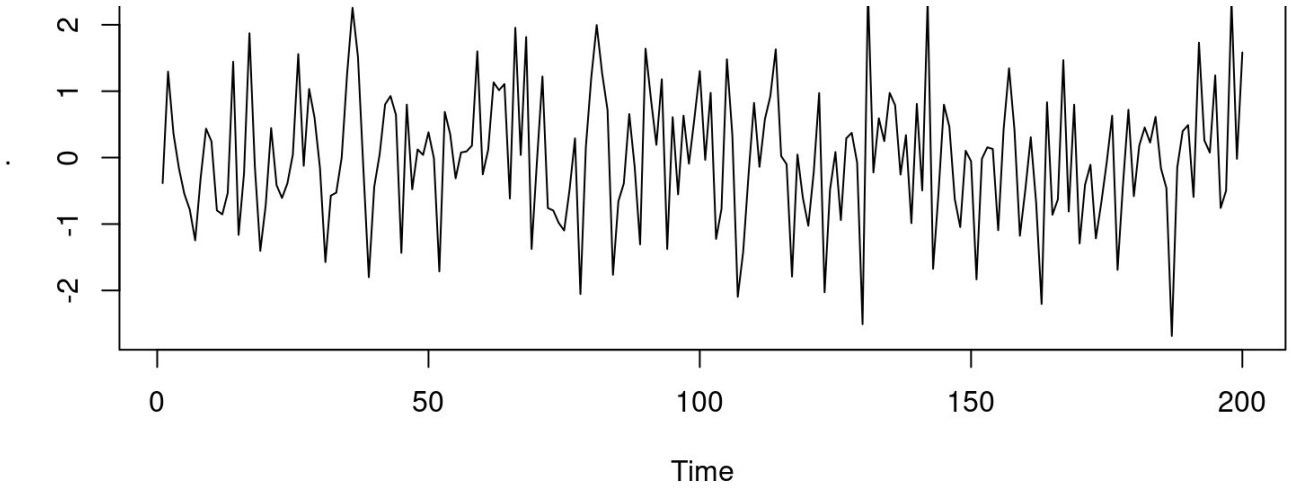
To fit a first-order moving average model to the data, where the estimation results are stored in the object `arma01` we execute the commands:

```
arma01 <- dat %>%
  pull(ma1) %>%
  arima(., order=c(0,0,1)) # uses ARIMA(p,d,q) with constant
arma01
```

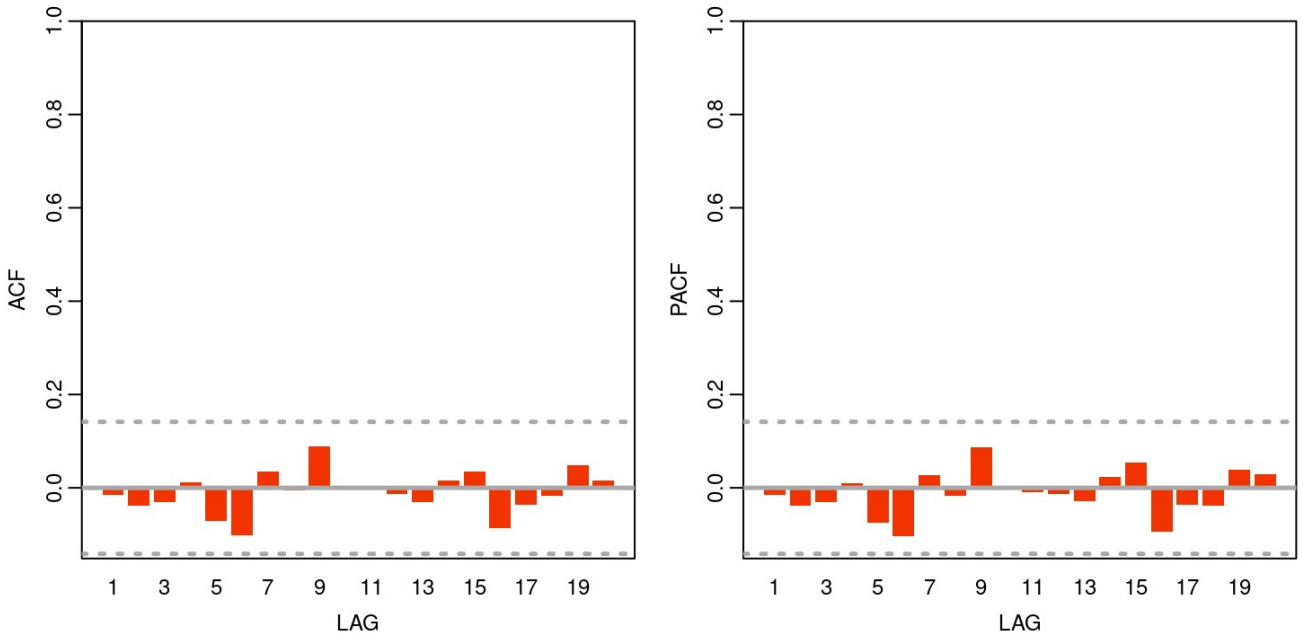
```
##
## Call:
## arima(x = ., order = c(0, 0, 1))
##
## Coefficients:
##          ma1  intercept
##       0.6945     0.0707
## s.e.  0.0550     0.1190
##
## sigma^2 estimated as 0.9896:  log likelihood = -283.07,  aic = 572.15
```

To inspect the residuals we execute the following commands and would note that they take on features of white noise.

```
par(mfrow=c(1, 1))
arma01$residuals %>%
  plot()
```



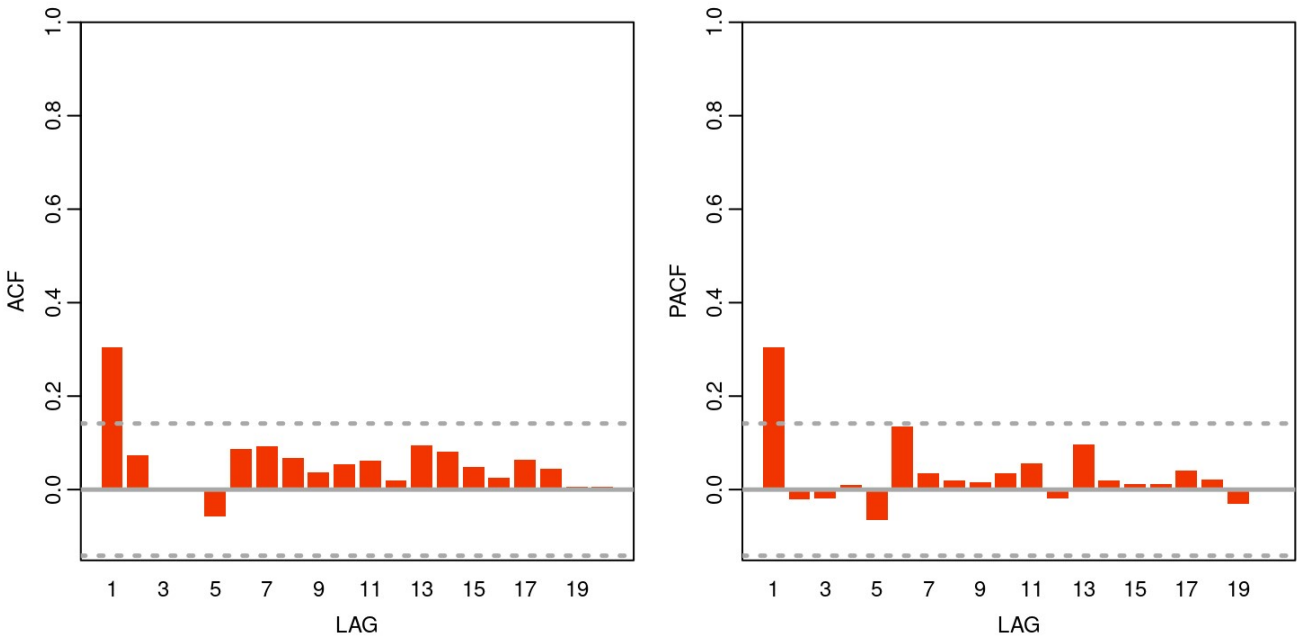

```
par(mfrow=c(1, 1))
arma01$residuals %>%
  ac(., max.lag=20)
```



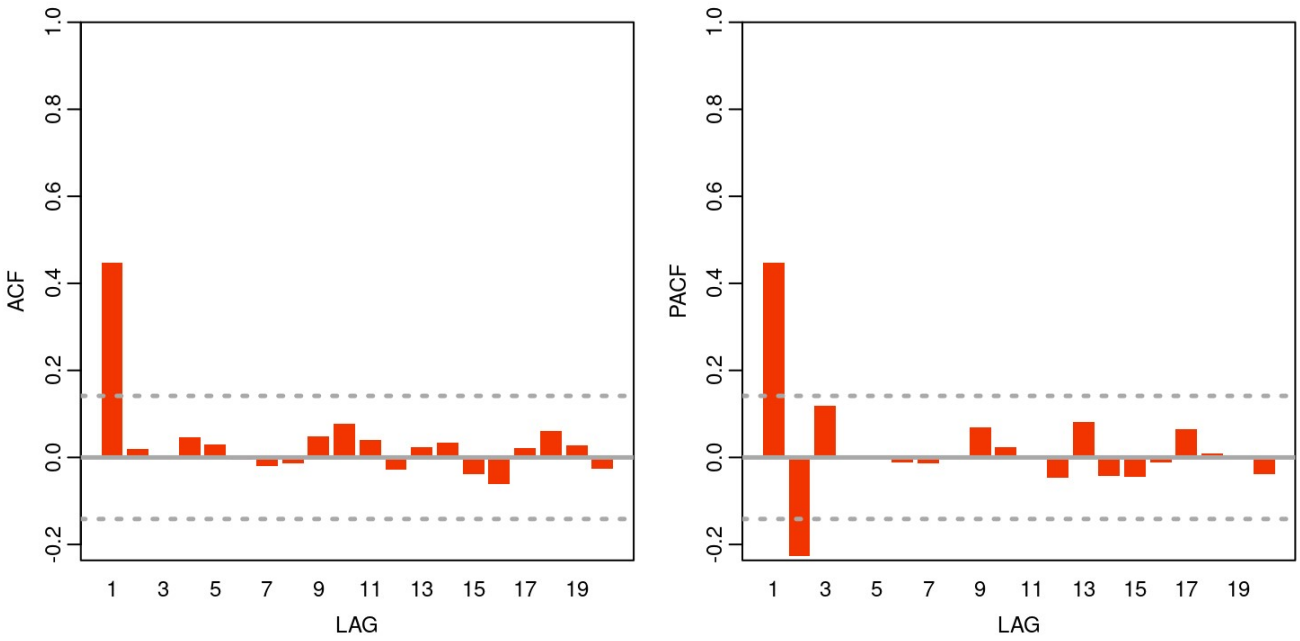
5 Simulated ARMA process

As noted in the lectures, the values of autocorrelation and partial autocorrelation functions for an ARMA process is equivalent to some form of weighted sum of these functions for the individual autoregressive and moving average components. This is displayed below, where we use the data that makes used of simulated autoregressive and moving average components before we provide the results of the ARMA process.

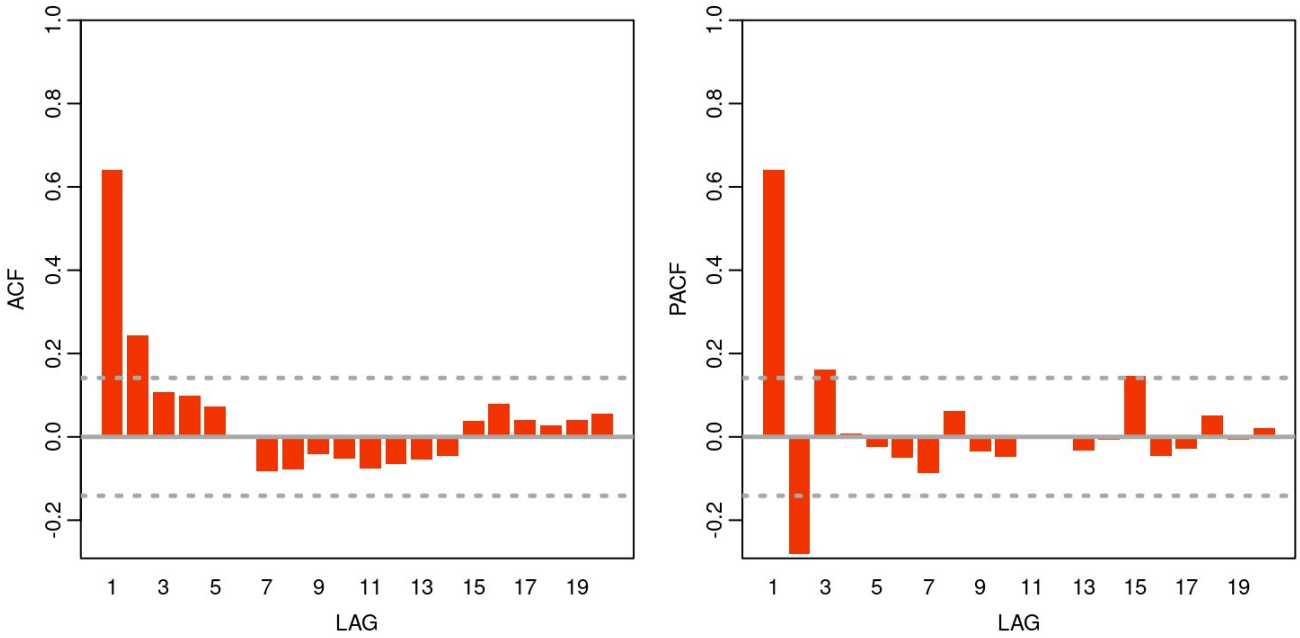
```
# autocorrelation function for ARMA(1,0)
dat %>%
  pull(arma10) %>%
  ac(., max.lag=20)
```




```
# autocorrelation function for ARMA(0,1)
dat %>%
  pull(arma01) %>%
  ac(., max.lag=20)
```

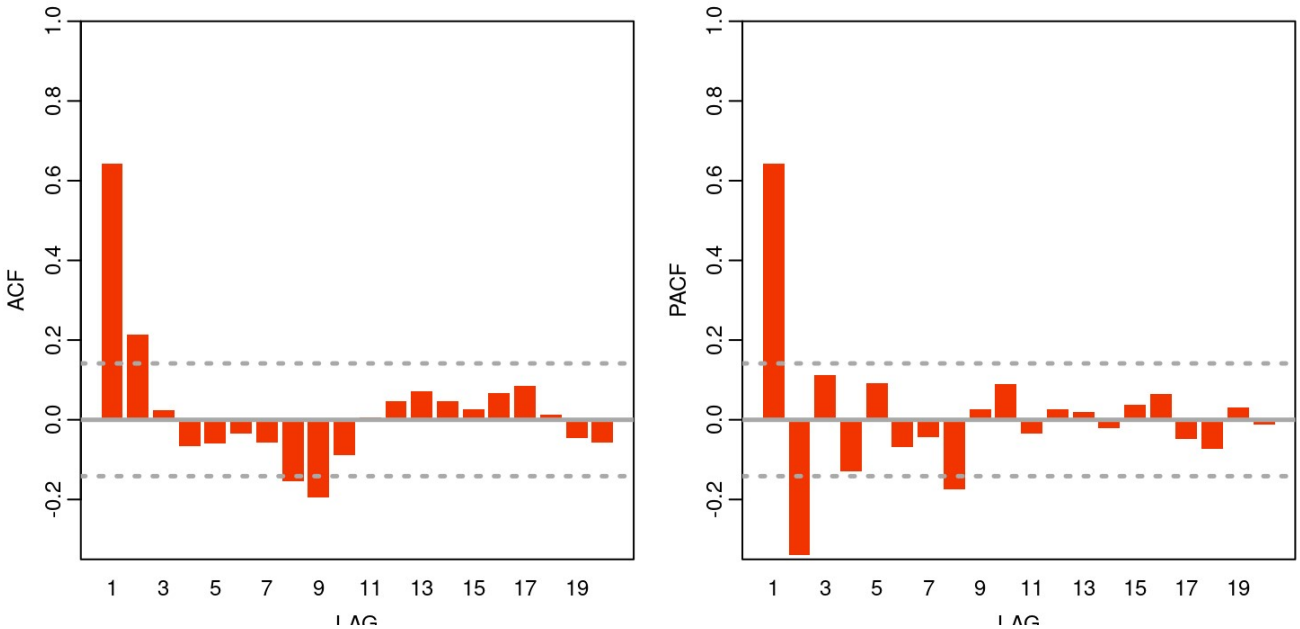


```
# autocorrelation function for ARMA(1,1)
dat %>%
  pull(arma11) %>%
  ac(., max.lag=20)
```



In the following exercise we will make use of the simulated ARMA(2,1) process and try to see whether we can identify it without any prior knowledge. To identify the process we will make use of the ACF & PACF as well as the information criteria.

```
dat %>%
  pull(arma21) %>%
  ac(., max.lag=20)
```



The results from the ACF & PACF would suggest that we are at most dealing with an ARMA(3,2). To estimate all models that may have an order that is equal to or less than order we could proceed as follows, before storing the AIC value in the object `arma.res`.

```
arma_res <- rep(0,16)
arma_res[1] <- arima(dat$arma21, order=c(3,0,2))$aic # fit arma(3,2) and save aic value
arma_res[2] <- arima(dat$arma21, order=c(2,0,2))$aic
arma_res[3] <- arima(dat$arma21, order=c(2,0,1))$aic
arma_res[4] <- arima(dat$arma21, order=c(1,0,2))$aic
arma_res[5] <- arima(dat$arma21, order=c(1,0,1))$aic
arma_res[6] <- arima(dat$arma21, order=c(3,0,0))$aic
arma_res[7] <- arima(dat$arma21, order=c(2,0,0))$aic
arma_res[8] <- arima(dat$arma21, order=c(0,0,2))$aic
arma_res[9]  <- arima(dat$arma21, order=c(3,0,2), include.mean=FALSE)$aic
arma_res[10] <- arima(dat$arma21, order=c(2,0,2), include.mean=FALSE)$aic
arma_res[11] <- arima(dat$arma21, order=c(2,0,1), include.mean=FALSE)$aic
arma_res[12] <- arima(dat$arma21, order=c(1,0,2), include.mean=FALSE)$aic
arma_res[13] <- arima(dat$arma21, order=c(1,0,1), include.mean=FALSE)$aic
arma_res[14] <- arima(dat$arma21, order=c(3,0,0), include.mean=FALSE)$aic
arma_res[15] <- arima(dat$arma21, order=c(2,0,0), include.mean=FALSE)$aic
arma_res[16] <- arima(dat$arma21, order=c(0,0,2), include.mean=FALSE)$aic
```

To find the model that has the lowest value for the AIC statistic we could execute the code:

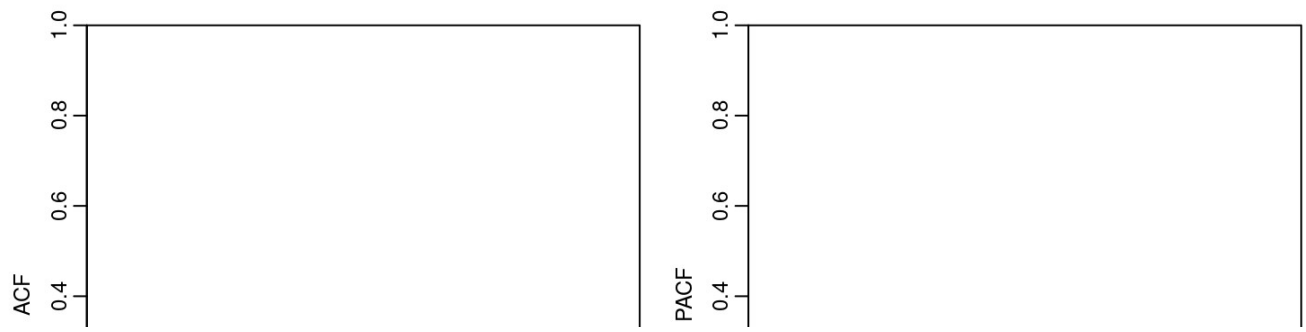
```
which(arma_res == min(arma_res))
```

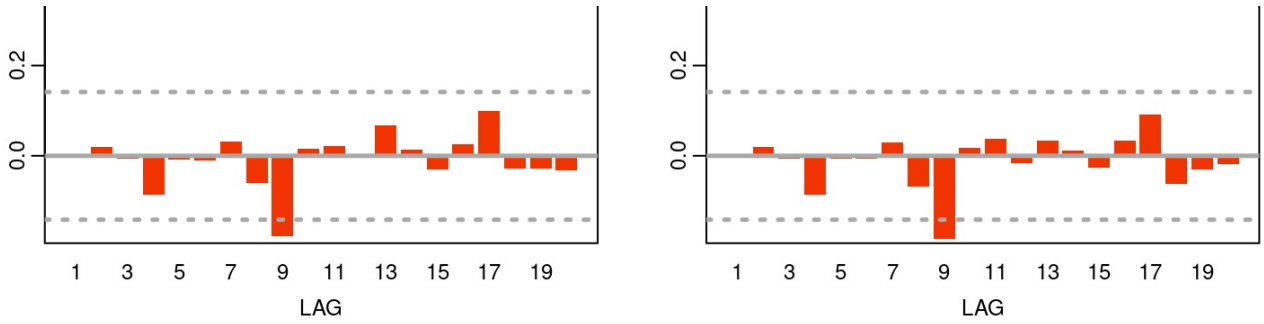
```
## [1] 13
```

This would suggest that ARMA(1,1) would provide a suitable fit for the model, which is perfect, but lets have a look at the dianostics. Hence, we re-estimate the model, store all the results and consider whether there is serial correlation in the residuals.

```
arma11 <- arima(dat$arma21, order=c(1, 0, 1), include.mean=FALSE)

arma11$residuals %>%
  ac(., max.lag=20)
```

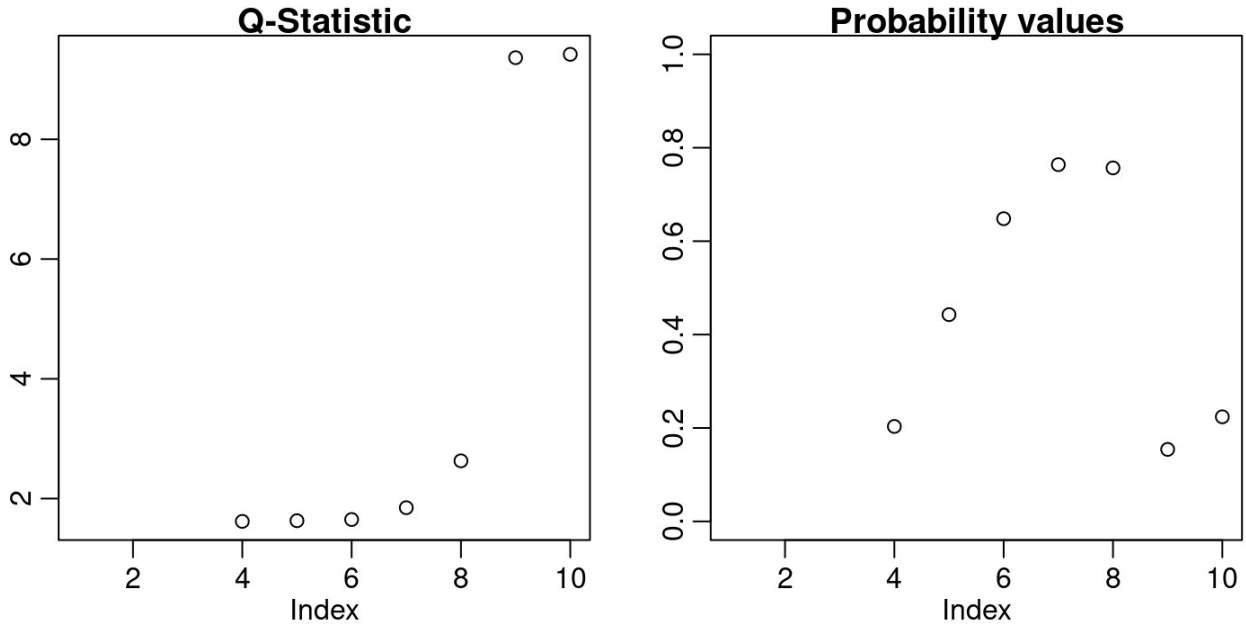





```
Q_stat <- rep(NA,10) # vector of ten observations
Q_prob <- rep(NA,10)

for (i in 4:10) {
  Q_stat[i] <- Box.test(arma11$residuals, lag=i, type="Ljung-Box", fitdf=3)$statistic
  Q_prob[i] <- Box.test(arma11$residuals, lag=i, type="Ljung-Box", fitdf=3)$p.value
}

op <- par(mfrow = c(1, 2))
plot(Q_stat, ylab = "", main='Q-Statistic')
plot(Q_prob, ylab = "", ylim=c(0,1), main='Probability values')
```




```
par(op)
```

This would appear to provide suitable results.

6 Univariate models for real data

6.1 South African gross domestic product

Before we work through the next part of this tutorial we will clear the workspace environment again and close all the figures that we have plotted.

```
rm(list=ls())
graphics.off()
```

In this tutorial we would like to make use of the following packages so we run the commands:

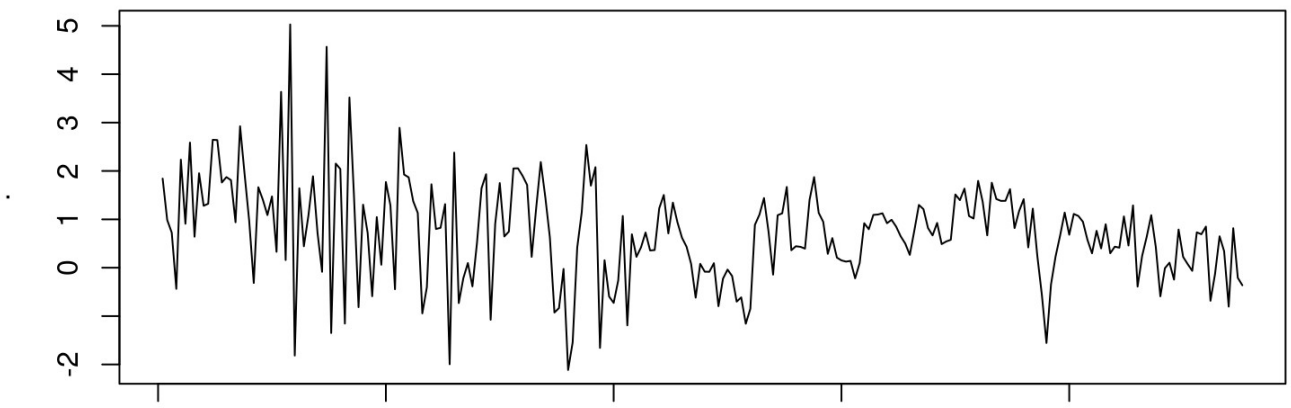
```
library(fArma)
library(forecast)
library(strucchange)
library(tidyverse)
library(lubridate)
library(tsm)
library(sarb2020q1)
```

As per the previous session we are going to make use of South African Real Gross Domestic Product data that is published by the South African Reserve Bank, which has the code KBP6006D. Hence, to retrieve the data for a particular release, which in this case 2020q1, we use the `sarb2020q1` package along with the following `tidyverse` functions:

```
gdp <- sarb_quarter %>%
  select(date, KBP6006D) %>%
  mutate(growth = 100 * ((KBP6006D / lag(KBP6006D)) - 1),
         grow_lag = lag(growth)) %>%
  drop_na()
```

To plot the series for GDP growth we can use the following commands.

```
gdp %>%
  pull(growth) %>%
  plot.ts()
```

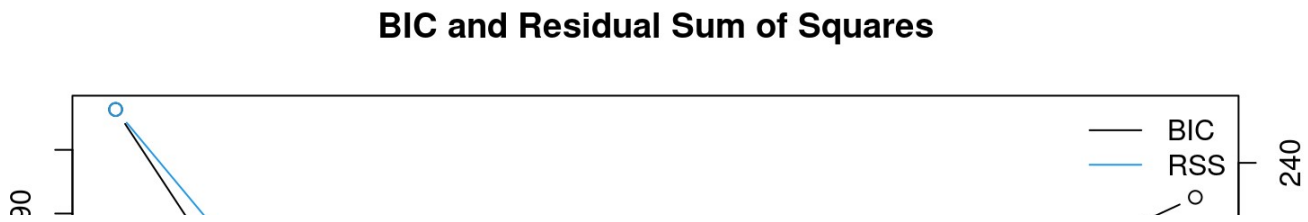


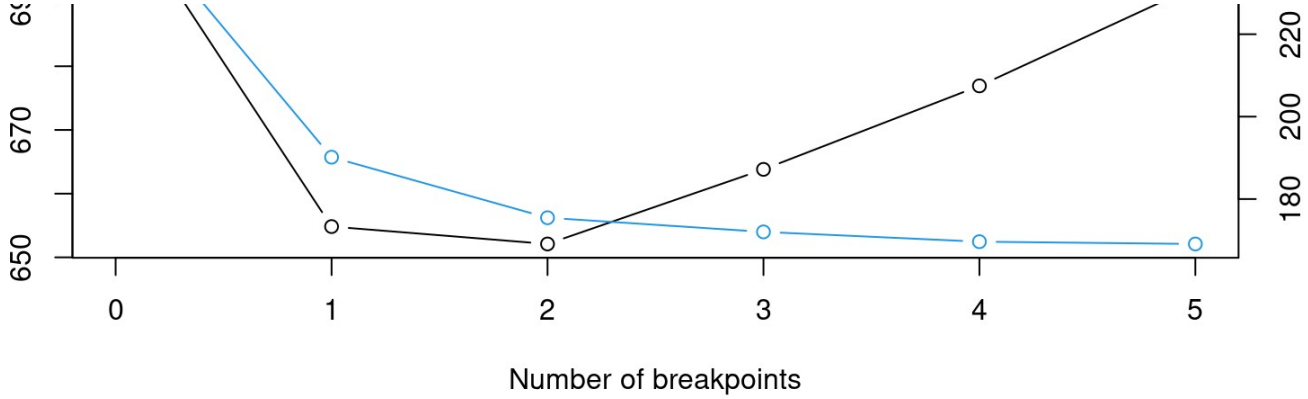
If we are of the opinion that this seems reasonable, we can proceed by checking for structural breaks. In this instance, we once again make use of an AR(1) model and the Bai and Perron (2003) statistic:

```
sa_bp <- breakpoints(growth ~ grow_lag, data = gdp, breaks = 5)
summary(sa_bp)  # where the breakpoints are
```

```
##
##   Optimal (m+1)-segment partition:
##
## Call:
## breakpoints.formula(formula = growth ~ grow_lag, breaks = 5,
##     data = gdp)
##
## Breakpoints at observation number:
##
## m = 1   42
## m = 2   42 78
## m = 3   42 78         203
## m = 4   42 78 130     203
## m = 5   42 78 130 166 203
##
## Corresponding to breakdates:
##
## m = 1   0.176470588235294
## m = 2   0.176470588235294 0.327731092436975
## m = 3   0.176470588235294 0.327731092436975
## m = 4   0.176470588235294 0.327731092436975
## m = 5   0.176470588235294 0.327731092436975
##
## m = 1
## m = 2
## m = 3
## m = 4   0.546218487394958
## m = 5   0.546218487394958 0.697478991596639
##
## m = 1
## m = 2
## m = 3   0.852941176470588
## m = 4   0.852941176470588
## m = 5   0.852941176470588
##
## Fit:
##
## m   0     1     2     3     4     5
## RSS 253.0 190.2 175.5 172.0 169.6 169.1
## BIC 706.3 654.8 652.1 663.8 676.9 692.6
```

```
plot(sa_bp, breaks = 5)
```






```
gdp %>%
  slice(sa_bp$breakpoint)
```

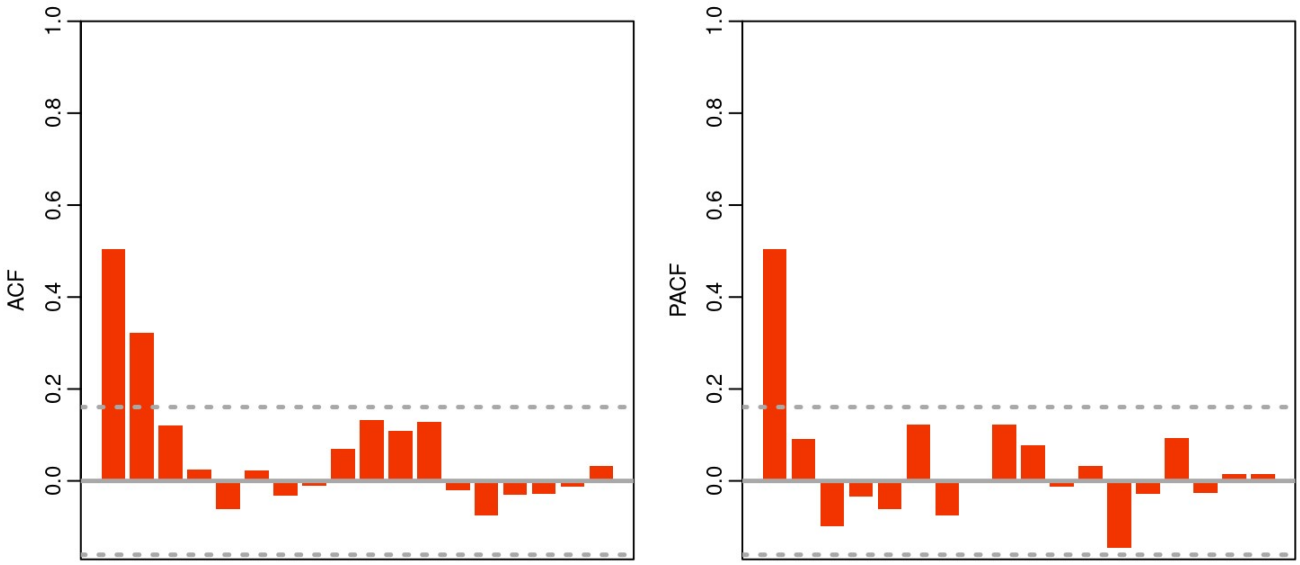
```
## # A tibble: 2 x 4
##   date       KBP6006D growth grow_lag
##   <date>        <dbl>  <dbl>    <dbl>
## 1 1970-10-01  1034117   3.52   -1.15
## 2 1979-10-01  1344927   2.05    0.746
```

As this suggests that there is a structural break around 1980, we make use of data from 1981. We will then label this subsample `y`.

```
y <- gdp %>%
  filter(date > ymd('1981-01-01')) %>%
  pull(growth)
```

To consider the degree of autocorrelation.

```
y %>% ac()
```



This variable has a bit of persistence and does not appear to be nonstationary. The maximum order of this variable is an AR(1), MA(2), or some combination of the two. We can then check the information criteria for a number of candidate models by constructing the following vector for the statistics:

```
arma_res <- rep(0,5)

arma_res[1] <- arima(y, order=c(1, 0, 2))$aic
arma_res[2] <- arima(y, order=c(1, 0, 1))$aic
arma_res[3] <- arima(y, order=c(1, 0, 0))$aic
arma_res[4] <- arima(y, order=c(0, 0, 2))$aic
arma_res[5] <- arima(y, order=c(0, 0, 1))$aic
```

To find the model with the smallest AIC value we can then execute the command:

```
which(arma_res == min(arma_res))
```

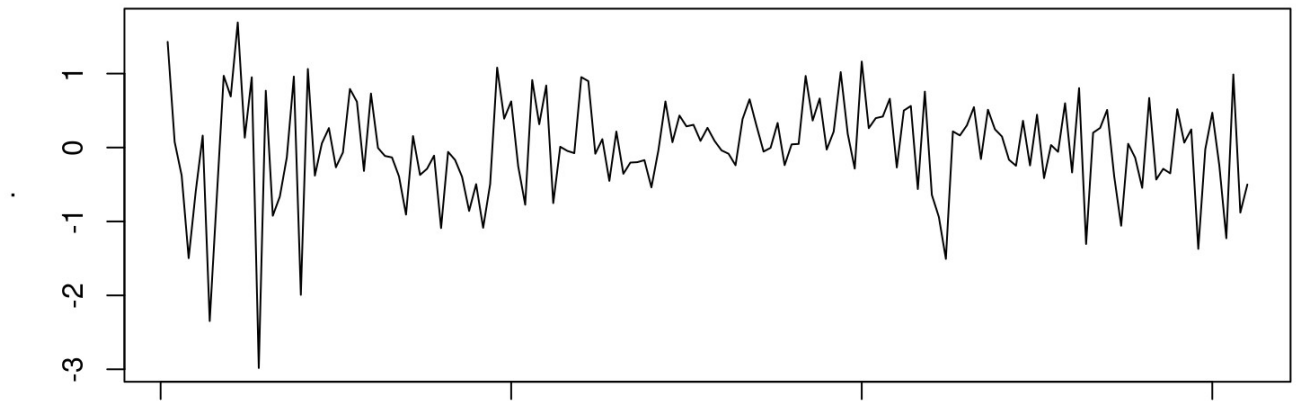
```
## [1] 3
```

These results suggest that the smallest value is provided by ARMA(1,0). With this in mind we estimate the parameter values for this model structure.

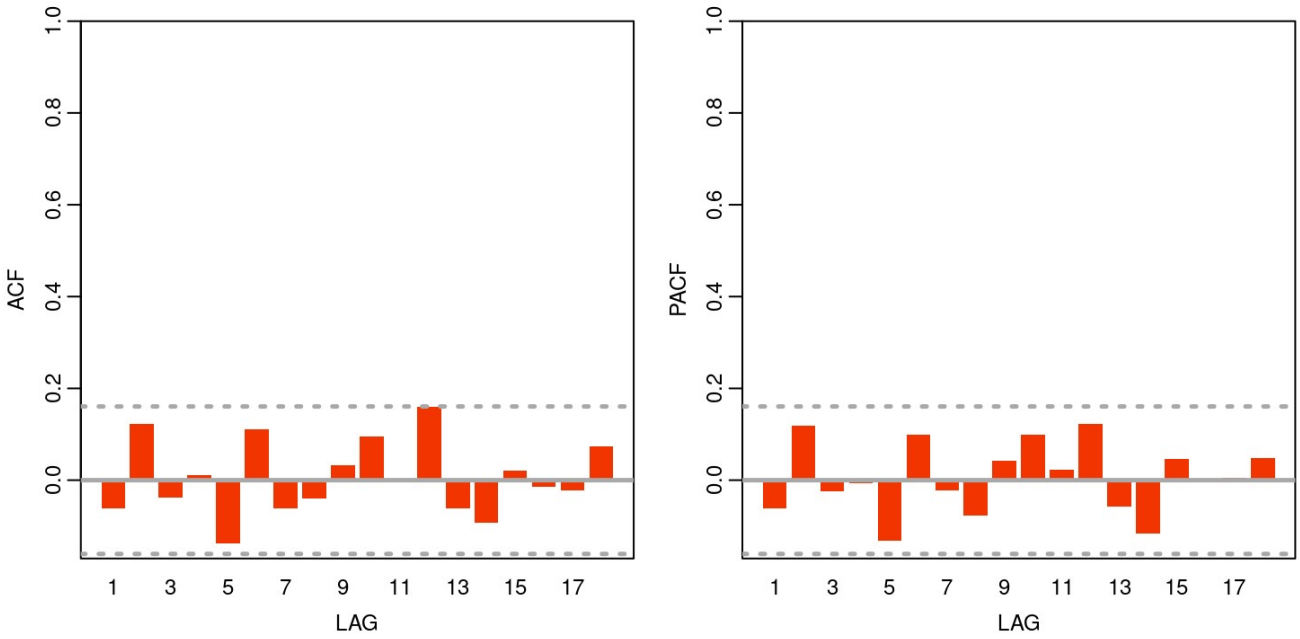
```
arma <- y %>%
  arima(., order=c(1,0,0))
```

Thereafter, we look at the residuals for the model to determine if there is any serial correlation.

```
par(mfrow=c(1, 1))
arma$residuals %>%
  plot()
```




```
arma$residuals %>%
  ac()
```

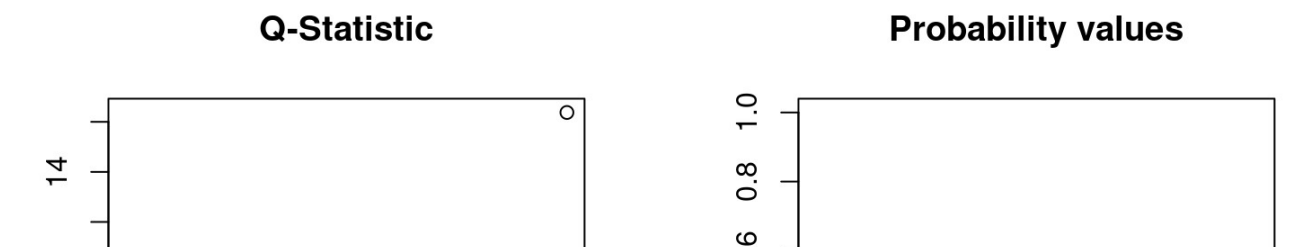


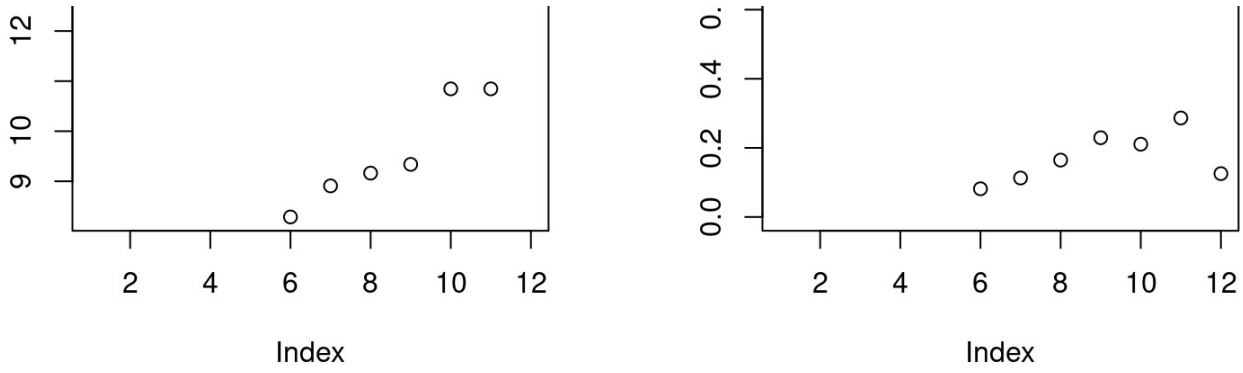
These results suggests that there is no remaining serial correlation, but we could also take a look at the Q-statistics to confirm this.

```
Q_stat <- rep(NA,12) # vector of twelve observations
Q_prob <- rep(NA,12)

for (i in 6:12) {
  Q_stat[i] <- Box.test(arma$residuals, lag = i, type =  "Ljung-Box",fitdf = 2)$statistic
  Q_prob[i] <- Box.test(arma$residuals, lag = i, type =  "Ljung-Box",fitdf = 2)$p.value
}

op <- par(mfrow = c(1, 2))
  plot(Q_stat, ylab = "", main='Q-Statistic')
  plot(Q_prob, ylab = "", ylim=c(0,1), main='Probability values')
```



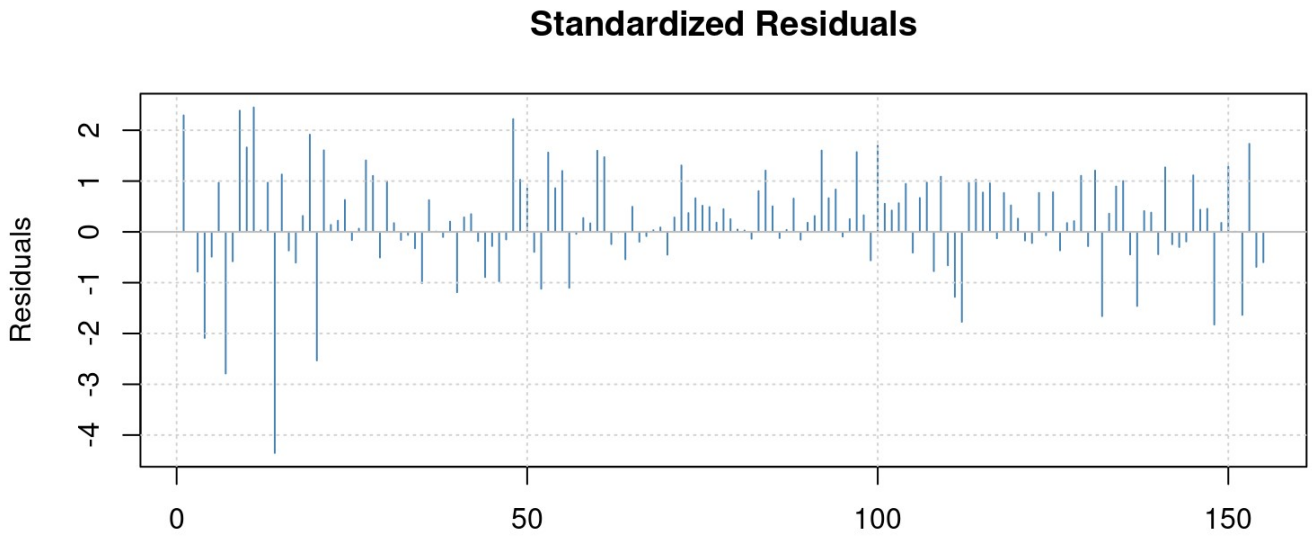


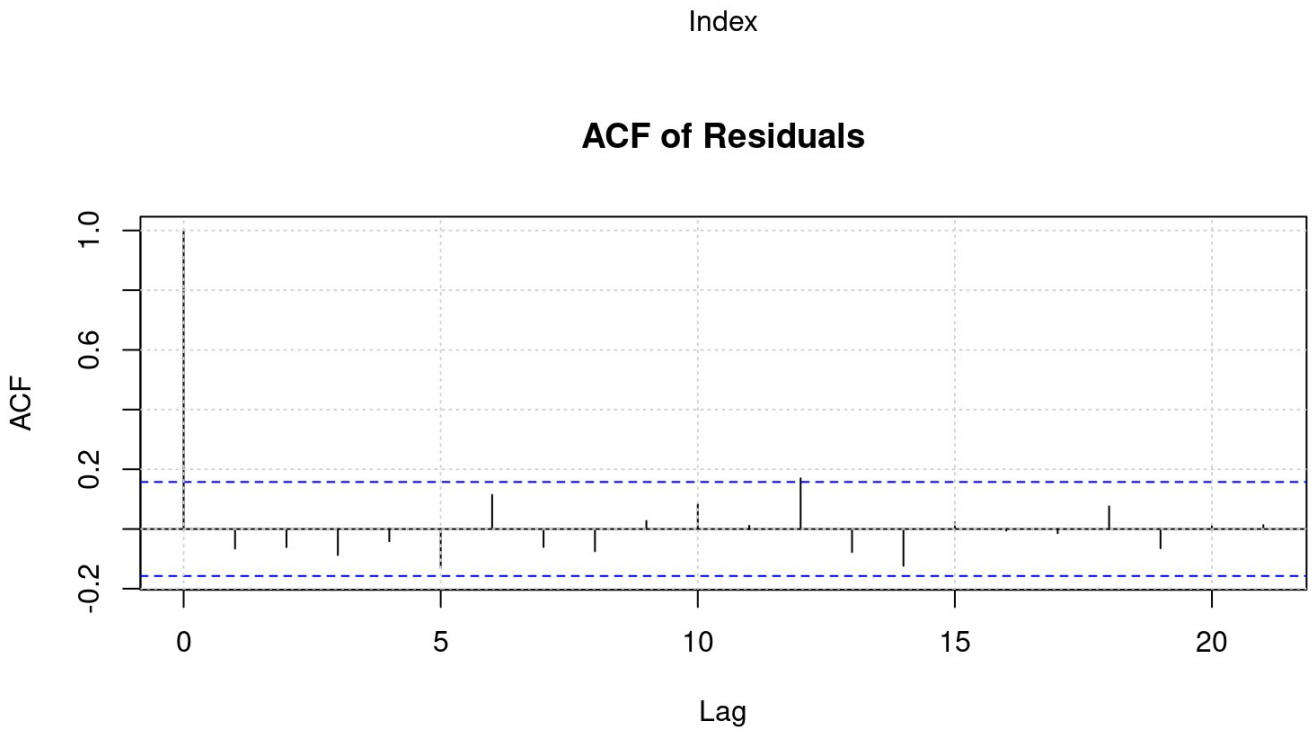

```
par(op)
```

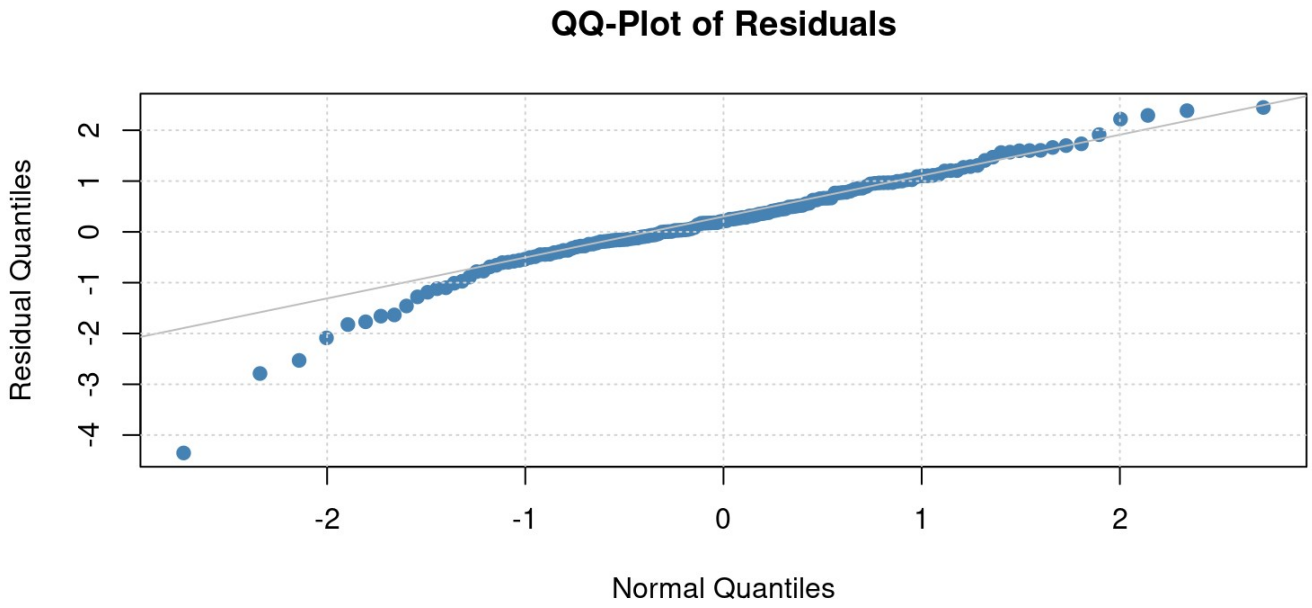
There certainly don't appear to be too many problems here and while this model would not be over-parameterised, we could have a look at what would happen if we tried to fit a over-parameterised model to the data. Recall that all the results from the model estimation are stored in the object `arma` and we could derive the *t*-statistics from the standard errors. However, there is another package, which is named `fArma` that displays these results in a more convenient manner, where we would need to estimate the model with the `armaFit` command.

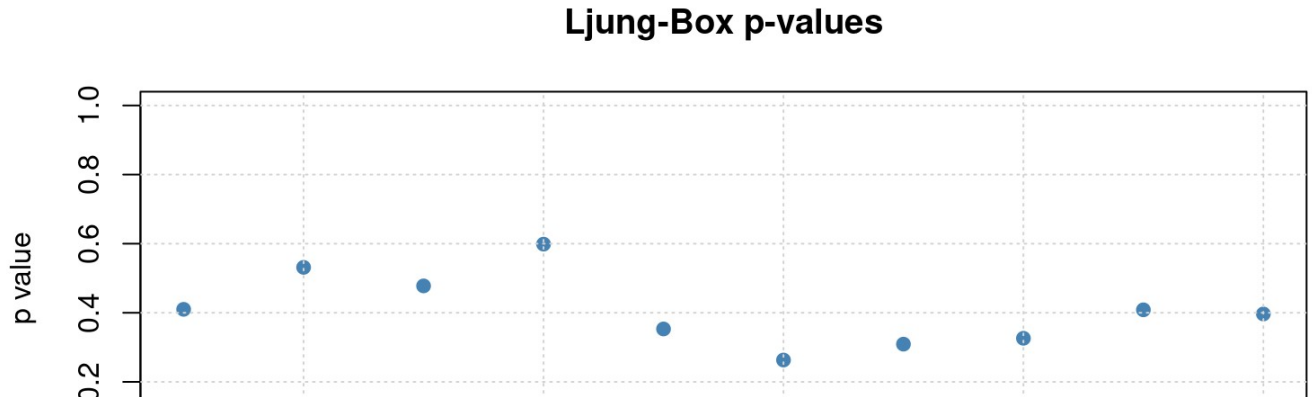
```
arma_fit1 <- fArma::armaFit(~ arma(1,2), data=y, include.mean=FALSE)
summary(arma_fit1)
```

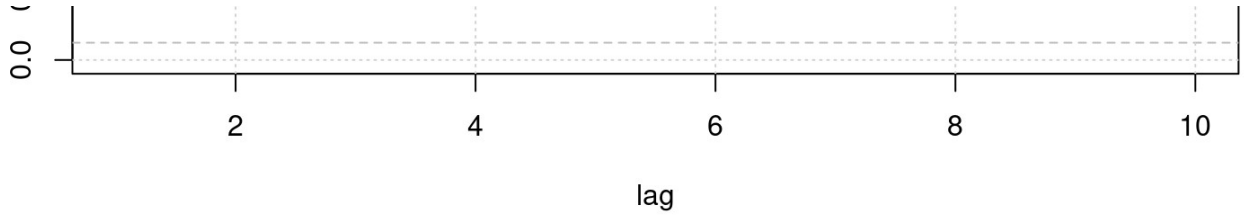
```
##
## Title:
##  ARIMA Modelling
##
## Call:
##  fArma::armaFit(formula = ~arma(1, 2), data = y, include.mean = FALSE)
##
## Model:
##  ARIMA(1,0,2) with method: CSS-ML
##
## Coefficient(s):
##     ar1      ma1      ma2
##  0.7192  -0.1588   0.1068
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0463 -0.1693  0.1480  0.5913  1.7136
##
## Moments:
## Skewness Kurtosis
##  -0.8399   2.7691
##
## Coefficient(s):
##      Estimate  Std. Error  t value Pr(>|t|)
## ar1    0.7192      0.1193    6.026 1.68e-09 ***
## ma1   -0.1588      0.1412   -1.125    0.261
## ma2    0.1068      0.1218    0.878    0.380
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## sigma^2 estimated as: 0.4899
## log likelihood:       -164.95
## AIC Criterion:        337.91
```




Index
ACF of Residuals
ACF
1.0
0.6
0.2
-0.2
0
5
10
15
20
Lag


QQ-Plot of Residuals
Residual Quantiles
2
1
0
-1
-2
-3
-4
-2
-1
0
1
2
Normal Quantiles


Ljung-Box p-values
p value
1.0
0.8
0.6
0.4
0.2

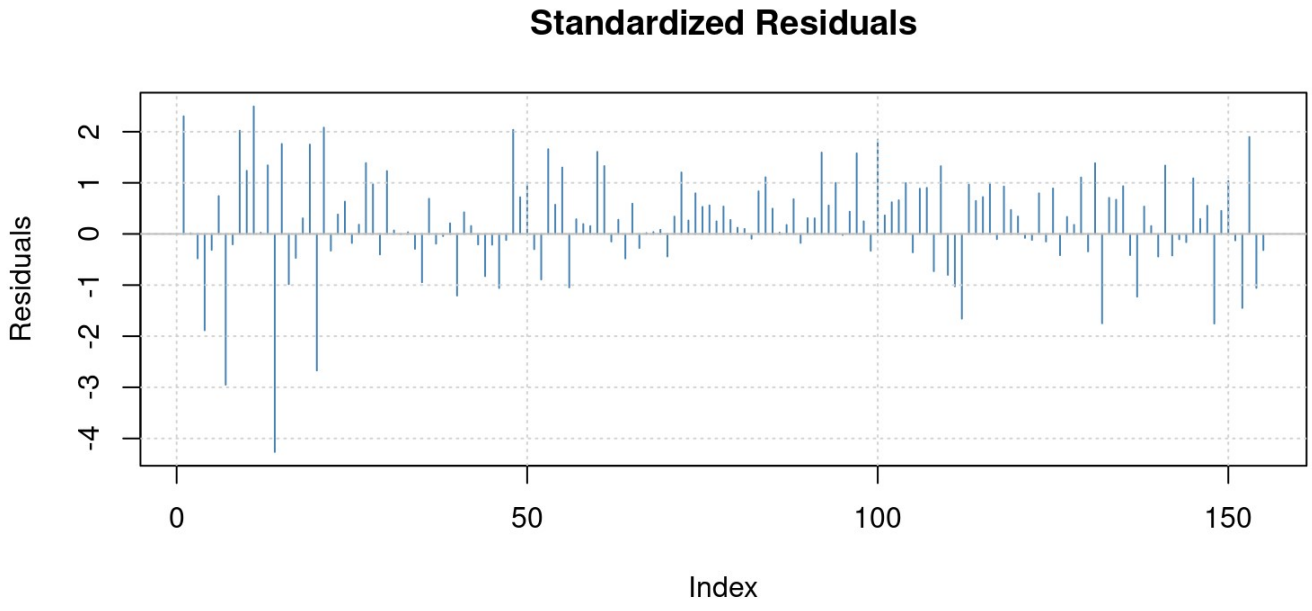


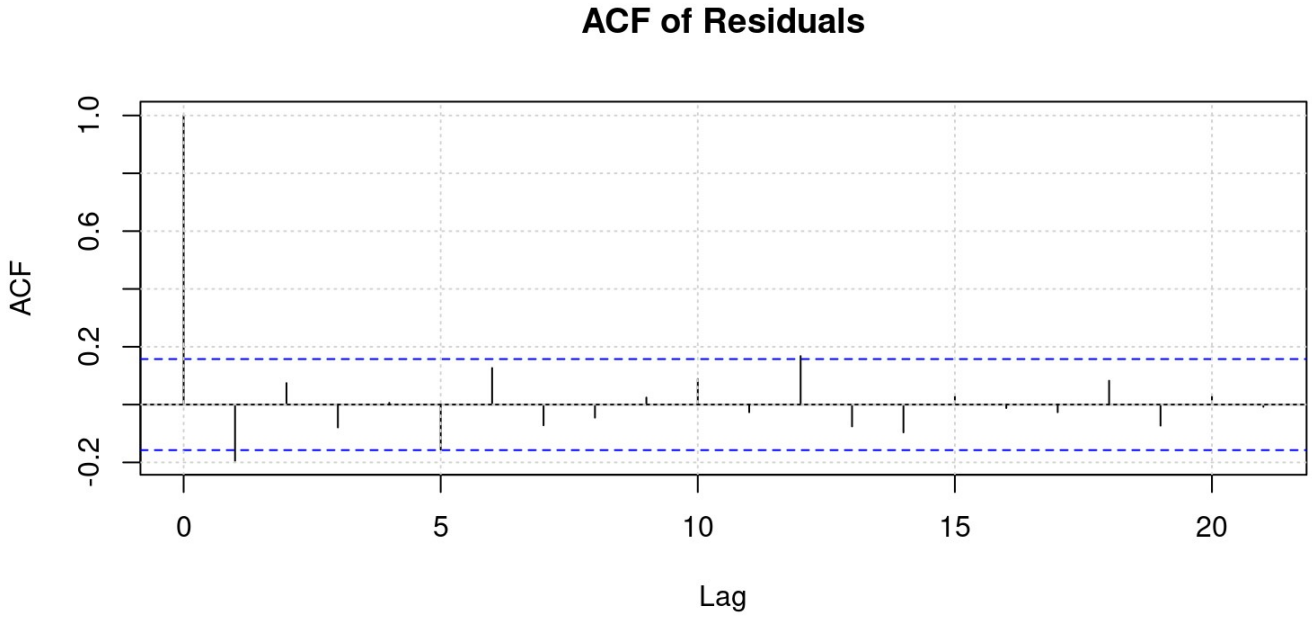

```
##
## Description:
##  Mon Aug 24 09:16:15 2020 by user: kevin
```

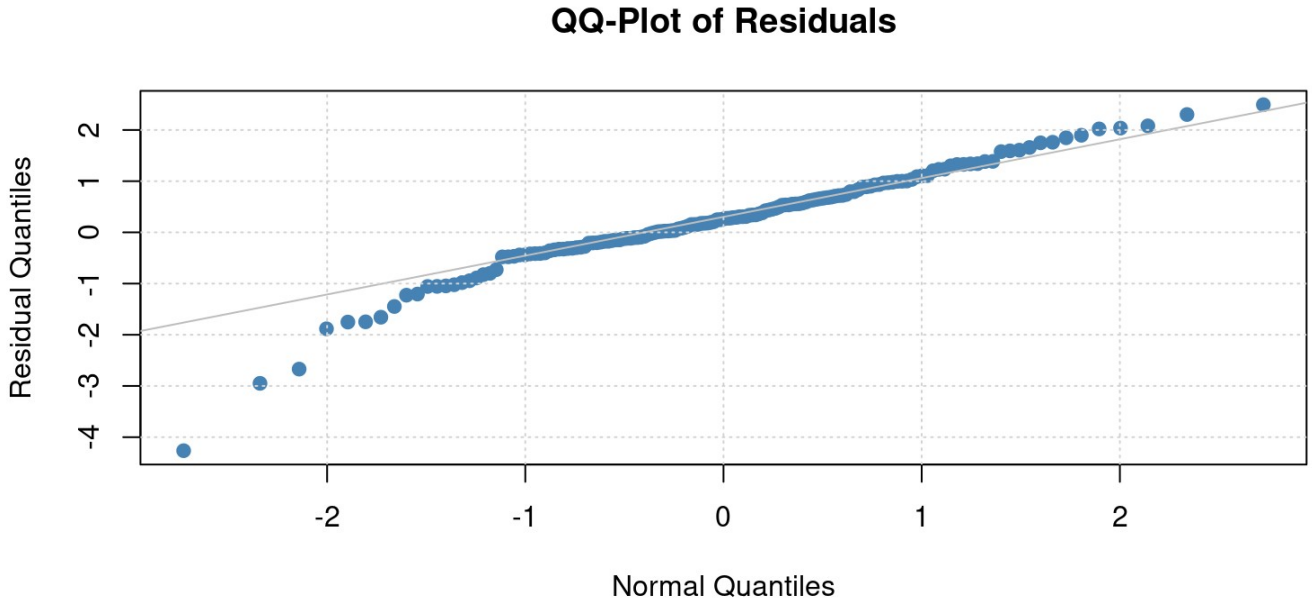
Note that the MA(1) and MA(2) parameters do not appear to be significant, so we should drop them and just estimate an AR(1) model.

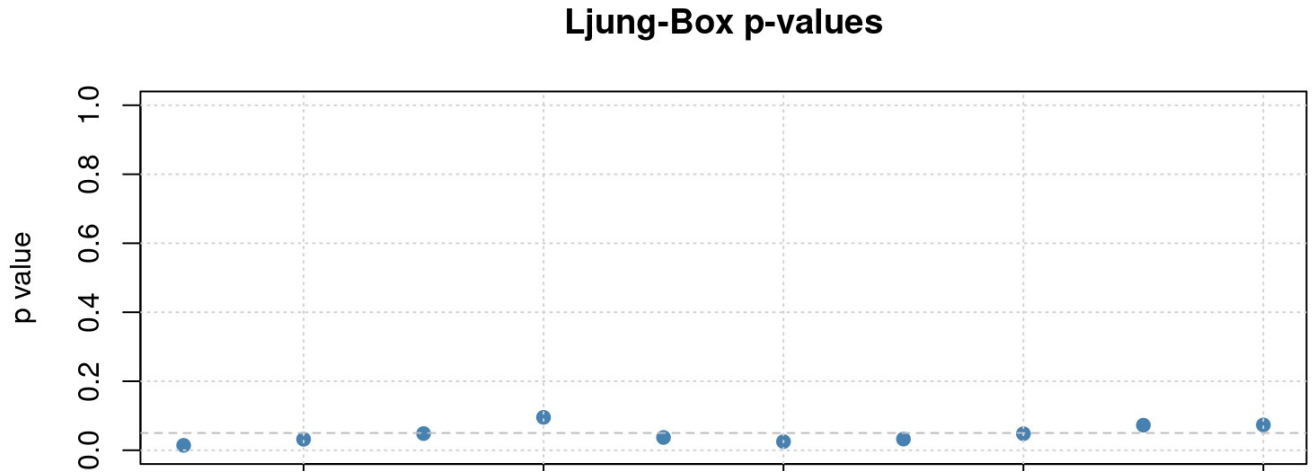
```
arma_fit2 <- fArma::armaFit(~ arma(1,0), data=y, include.mean=FALSE)
summary(arma_fit2)
```

```
##
## Title:
##  ARIMA Modelling
##
## Call:
##  fArma::armaFit(formula = ~arma(1, 0), data = y, include.mean = FALSE)
##
## Model:
##  ARIMA(1,0,0) with method: CSS-ML
##
## Coefficient(s):
##   ar1
## 0.662
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0327 -0.1479  0.1871  0.5792  1.7734
##
## Moments:
## Skewness Kurtosis
##  -0.8806   2.8581
##
## Coefficient(s):
##      Estimate  Std. Error  t value Pr(>|t|)
## ar1   0.66196     0.06128     10.8   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## sigma^2 estimated as: 0.5058
## log likelihood:       -167.39
## AIC Criterion:        338.78
```




ACF of Residuals
ACF
1.0
0.6
0.2
-0.2
0
5
10
15
20
Lag


QQ-Plot of Residuals
Residual Quantiles
2
1
0
-1
-2
-3
-4
-2
-1
0
1
2
Normal Quantiles


Ljung-Box p-values
p value
1.0
0.8
0.6
0.4
0.2
0.0


```
##
## Description:
##  Mon Aug 24 09:16:15 2020 by user: kevin
```

Another useful package is the `forecast` package, which you would know how to install by now. After you provide yourself with access to the routines in the package, you can utilise the `auto.arima` command, which allows you to specify the maximum number of autoregressive and moveing average lags. Thereafter, you can decide which information criteria should be used to select the final model and everything will be done for you.

```
auto_mod <- forecast::auto.arima(y, max.p = 2, max.q = 2, start.p = 0, start.q = 0,
                       stationary = TRUE, seasonal = FALSE, allowdrift = FALSE,
                       ic = "aic")

auto_mod # AR(1) with intercept
```

```
## Series: y
## ARIMA(1,0,0) with non-zero mean
##
## Coefficients:
##          ar1    mean
##       0.5191  0.5122
## s.e.  0.0698  0.1129
##
## sigma^2 estimated as 0.4688:  log likelihood=-160.38
## AIC=326.75   AICc=326.91   BIC=335.88
```

As the name suggests, this package has a number of useful forecasting routines, which is the subject of next weeks lecture. For example, to generate a forecast for the last 4 years of data, we remove the out-of-sample data from that which is going to be used for the in-sample estimation.

```
y_sub <- gdp %>%
  filter(date > ymd('1981-01-01')) %>%
  slice(1:(n() - 12)) %>%
  pull(growth)
```

In this case, we can compare the out-of-sample results for the ARIMA(1,2) with the AR(1) model.

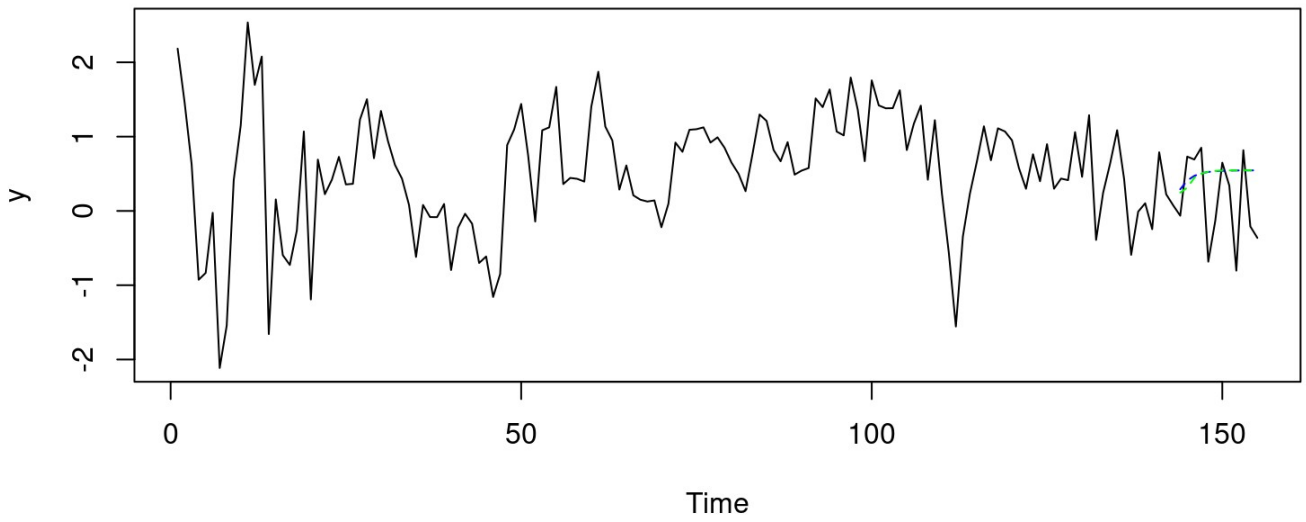
```
arma10 <- y_sub %>%
  arima(., order=c(1, 0, 0))
arma12 <- y_sub %>%
  arima(., order=c(1, 0, 2))
```

To forecast foreward we can use the predict command, as follows:

```
arma10_pred <- predict(arma10, n.ahead=12)
arma12_pred <- predict(arma12, n.ahead=12)
```

To plot these results we would use the `plot.ts` and `lines` commands that is used for displaying the results on the same set of axes.

```
par(mfrow=c(1, 1))
plot.ts(y)
lines(arma10_pred$pred, col="blue", lty=2)
lines(arma12_pred$pred, col="green", lty=2)
```



Unfortunately, these forecasting results look fairly poor.

6.2 South African consumer price index

Once again, we will start off this part of the tutorial by clearing the workspace environment and closing all the plots.

```
rm(list=ls())
graphics.off()
```

Once again we are going to make use of the the same packages:

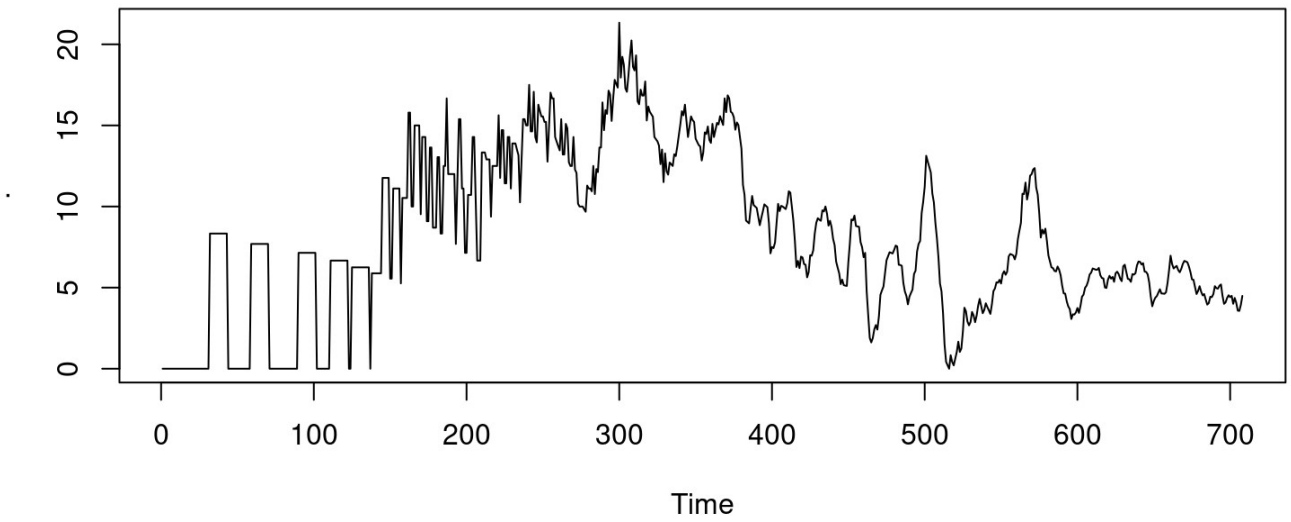
```
library(fArma)
library(forecast)
library(astsa)
library(strucchange)
library(tidyverse)
library(lubridate)
library(tsm)
library(sarb2020q1)
```

To access the CPI data in the `sarb2020q1` package, we would want to extract the series KBP7170N that is published by the South African Reserve Bank on a monthly basis. To create an object for this data before calculating the rate of year-on-year inflation, we can execute the following commands:

```
dat <- sarb_month %>%
  select(date, KBP7170N) %>%
  mutate(inf_yoy  = 100 * ((KBP7170N / lag(KBP7170N, n = 12)) - 1),
         infl_lag = lag(inf_yoy)) %>%
  drop_na()
```

We can then plot the data to make sure that it looks reasonable.

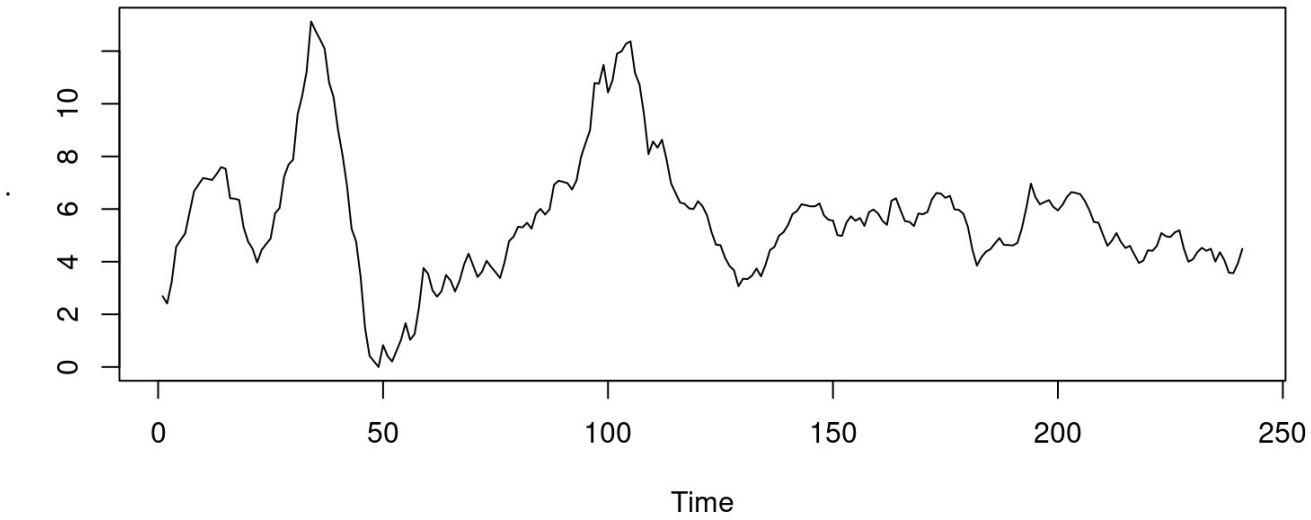
```
dat %>%
  pull(inf_yoy) %>%
  plot.ts()
```



The start of the series looks problematic, so let's consider data from January 2000 onwards.

```
dat <- dat %>%
  filter(date >= ymd("2000-01-01"))

dat %>%
  pull(inf_yoy) %>%
  plot.ts()
```

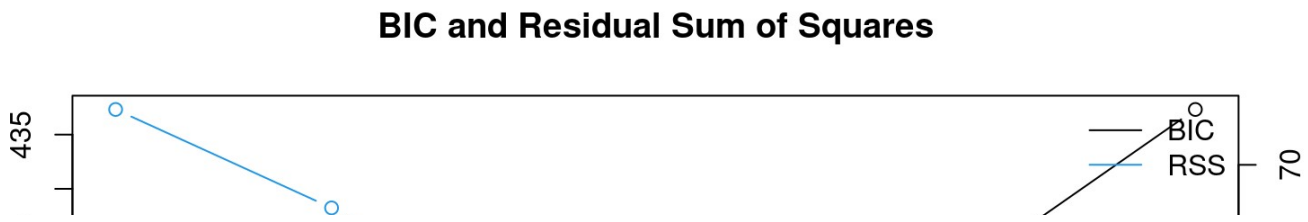


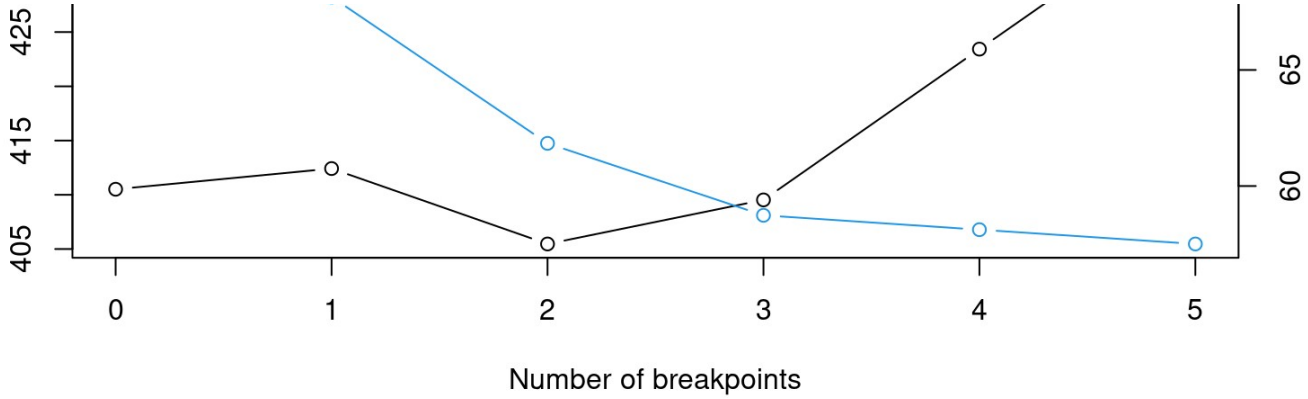
We can then proceed to check for structural breaks. In this instance, we once again make use of an AR(1) model, for which we have already created the lag of year-on-year inflation:

```
sa_bp <- breakpoints(inf_yoy ~ infl_lag, data = dat, breaks = 5)
summary(sa_bp)  # where the breakpoints are
```

```
##
##   Optimal (m+1)-segment partition:
##
## Call:
## breakpoints.formula(formula = inf_yoy ~ infl_lag, breaks = 5,
##     data = dat)
##
## Breakpoints at observation number:
##
## m = 1         105
## m = 2   37 76
## m = 3   37 76 112
## m = 4   37 76 112     205
## m = 5   37 76 112 152 205
##
## Corresponding to breakdates:
##
## m = 1
## m = 2   0.153526970954357 0.315352697095436
## m = 3   0.153526970954357 0.315352697095436
## m = 4   0.153526970954357 0.315352697095436
## m = 5   0.153526970954357 0.315352697095436
##
## m = 1   0.435684647302905
## m = 2
## m = 3   0.464730290456431
## m = 4   0.464730290456431
## m = 5   0.464730290456431 0.630705394190871
##
## m = 1
## m = 2
## m = 3
## m = 4   0.850622406639004
## m = 5   0.850622406639004
##
## Fit:
##
## m   0      1      2      3      4      5
## RSS  72.39  68.15  61.84  58.74  58.12  57.50
## BIC 410.51 412.42 405.47 409.53 423.43 437.32
```

```
plot(sa_bp, breaks = 5)
```






```
dat %>%
  slice(sa_bp$breakpoint)
```

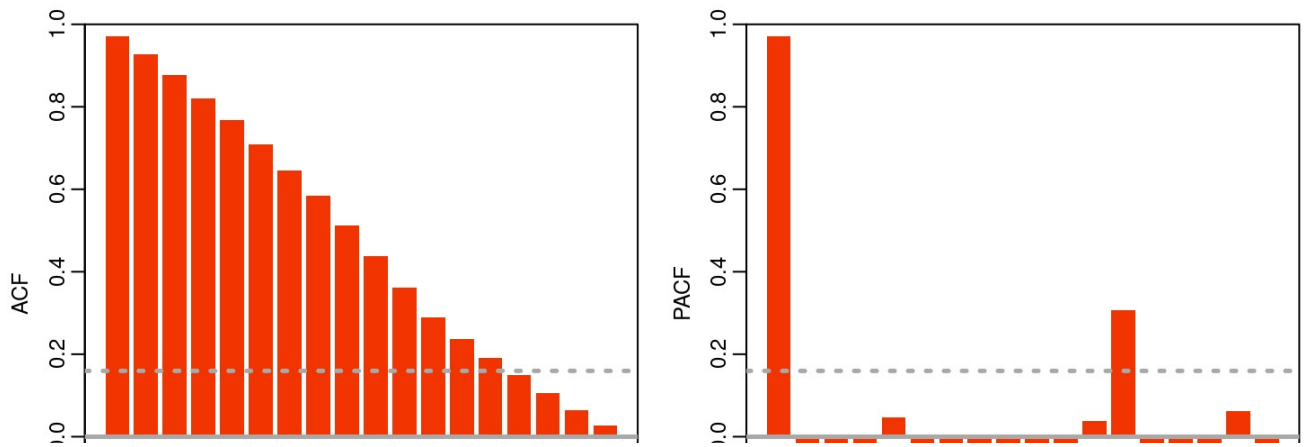
```
## # A tibble: 2 x 4
##   date       KBP7170N inf_yoy infl_lag
##   <date>        <dbl>   <dbl>    <dbl>
## 1 2003-01-01     48.2   12.1      12.4
## 2 2006-04-01     52      3.38      3.6
```

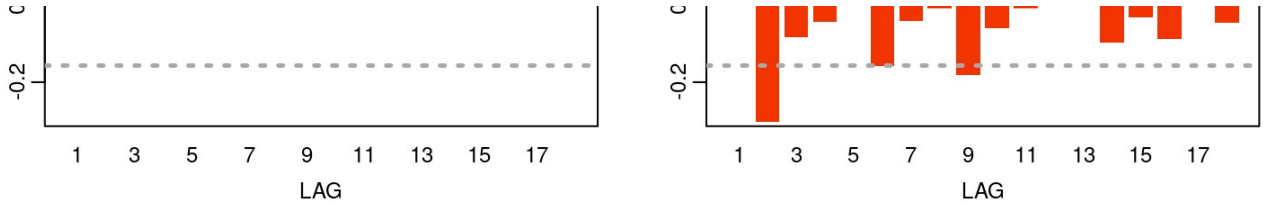
These results suggest that there could be a breakpoint during the year 2006, so let's take a subset of data from the start of 2007.

```
y <- dat %>%
  filter(date > ymd('2006-12-01')) %>%
  pull(inf_yoy)
```

We can now take a look at the persistence in the data, by plotting the autocorrelation function. Note that the first autocorrelation coefficient is around 0.97, which suggests that this could be a random-walk process. As such we should test for a unit root by using ADF (or similar) test, but as we are only going to cover this topic in a later lecture, we are going to proceed as if the process is stationary.

```
y %>% ac()
```





If we allow for a maximum order for an AR(2) and/or an MA(4) - although the moving average component could be of a higher order, we can then check the information criteria for a number of candidate models by constructing the following vector for the statistics:

```
arma_res <- rep(0,8)
arma_res[1]  <- arima(y, order=c(2, 0, 4))$aic
arma_res[2]  <- arima(y, order=c(2, 0, 3))$aic
arma_res[3]  <- arima(y, order=c(2, 0, 2))$aic
arma_res[4]  <- arima(y, order=c(2, 0, 1))$aic
arma_res[5]  <- arima(y, order=c(2, 0, 0))$aic
arma_res[6]  <- arima(y, order=c(1, 0, 2))$aic
arma_res[7]  <- arima(y, order=c(1, 0, 1))$aic
arma_res[8] <- arima(y, order=c(1, 0, 0))$aic
```

To find the model with the smallest AIC value we can then execute the command:

```
which(arma_res == min(arma_res))
```

```
## [1] 2
```

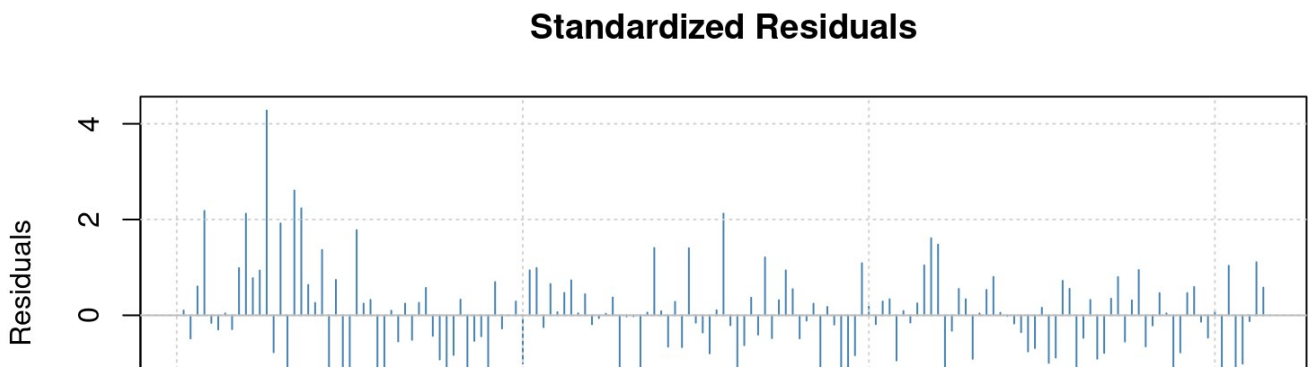
These results suggest that the smallest value is provided by ARMA(2,3). With this in mind we estimate the parameter values for this model structure.

```
arma <- y %>%
  arima(., order=c(2,0,3))
```

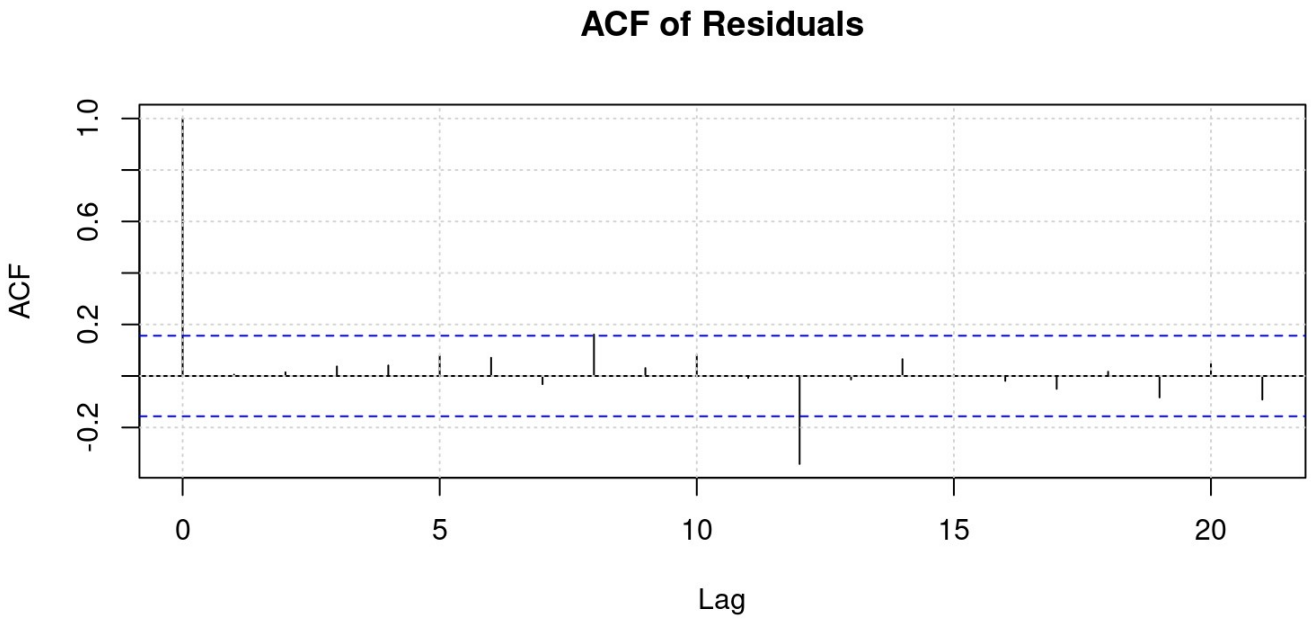
Let's make sure that the coefficients are significant

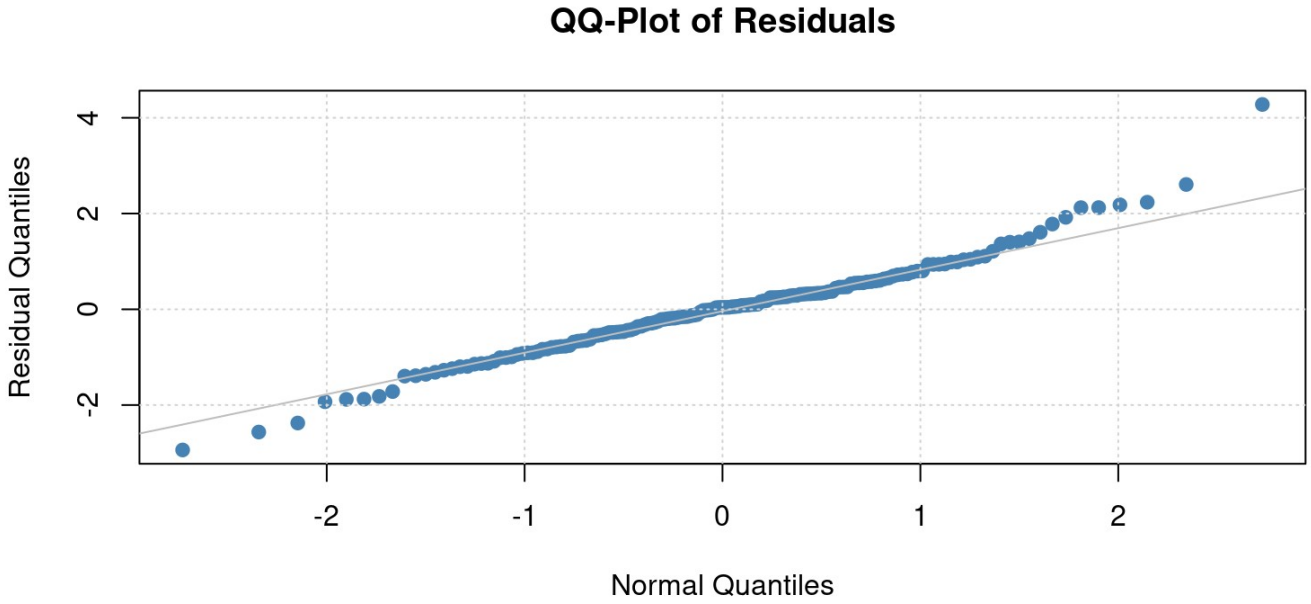
```
arma_fit <- fArma::armaFit(~ arma(2,3), data=y)
summary(arma_fit)
```

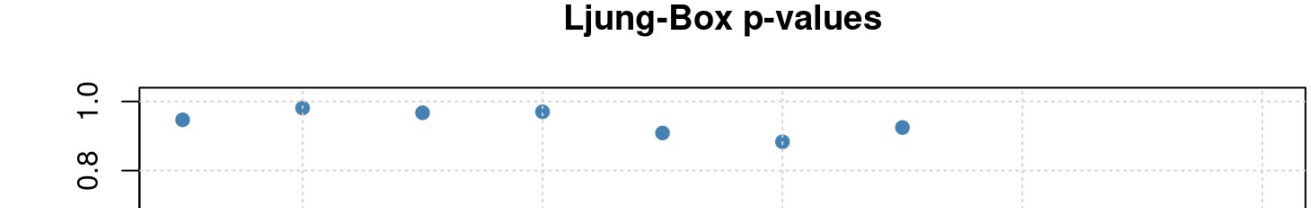
```
##
## Title:
##  ARIMA Modelling
##
## Call:
##  fArma::armaFit(formula = ~arma(2, 3), data = y)
##
## Model:
##  ARIMA(2,0,3) with method: CSS-ML
##
## Coefficient(s):
##       ar1        ar2        ma1        ma2        ma3
##    0.1083     0.7893     1.1821     0.4017     0.2196
## intercept
##    5.8240
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1817 -0.2511  0.0163  0.2197  1.7206
##
## Moments:
## Skewness Kurtosis
##   0.3834   1.9153
##
## Coefficient(s):
##             Estimate  Std. Error  t value Pr(>|t|)
## ar1          0.10825     0.06488    1.669  0.09520 .
## ar2          0.78932     0.06494   12.155  < 2e-16 ***
## ma1          1.18207     0.09566   12.357  < 2e-16 ***
## ma2          0.40166     0.12895    3.115  0.00184 **
## ma3          0.21957     0.09269    2.369  0.01784 *
## intercept    5.82403     0.79650    7.312 2.63e-13 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## sigma^2 estimated as: 0.1617
## log likelihood:       -82.65
## AIC Criterion:        179.29
```

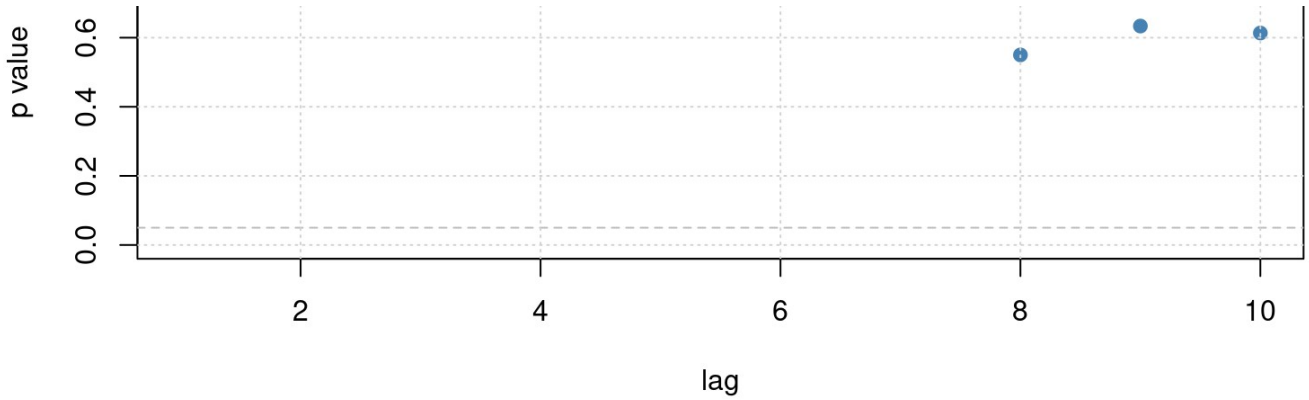



-2
0
50
100
150
Index


ACF of Residuals
ACF
1.0
0.6
0.2
-0.2
0
5
10
15
20
Lag


QQ-Plot of Residuals
Residual Quantiles
4
2
0
-2
-2
-1
0
1
2
Normal Quantiles

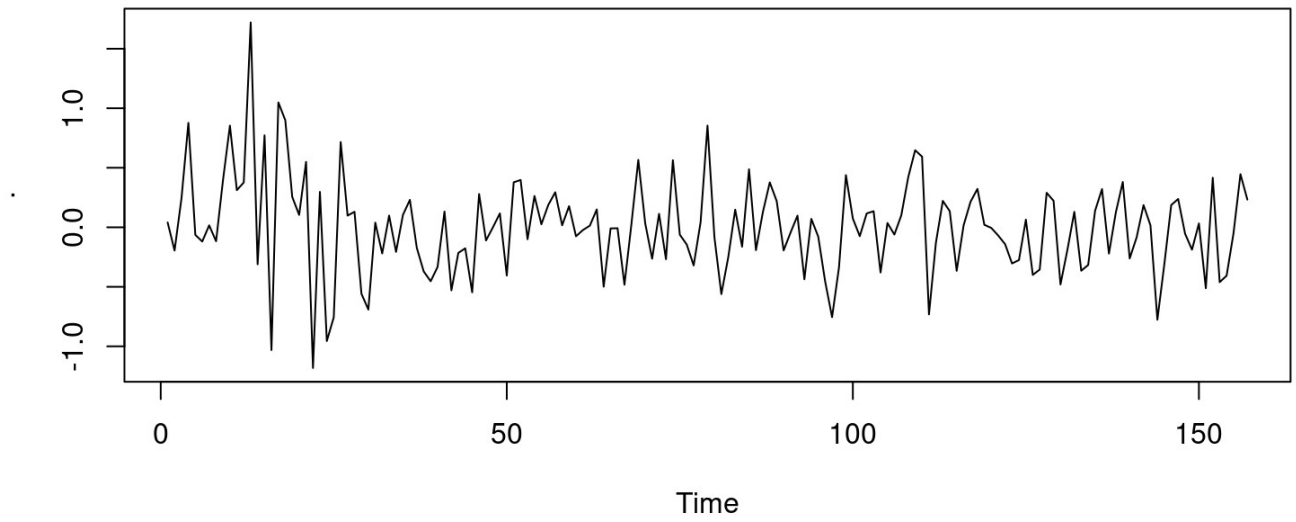

Ljung-Box p-values
1.0
0.8



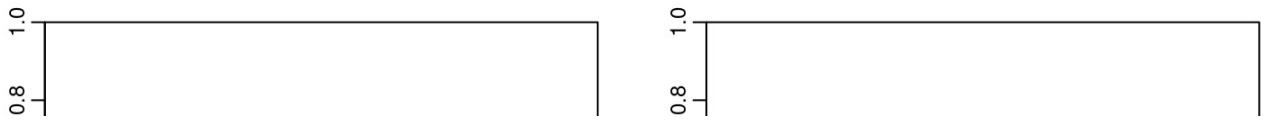

```
##
## Description:
##  Mon Aug 24 09:16:17 2020 by user: kevin
```

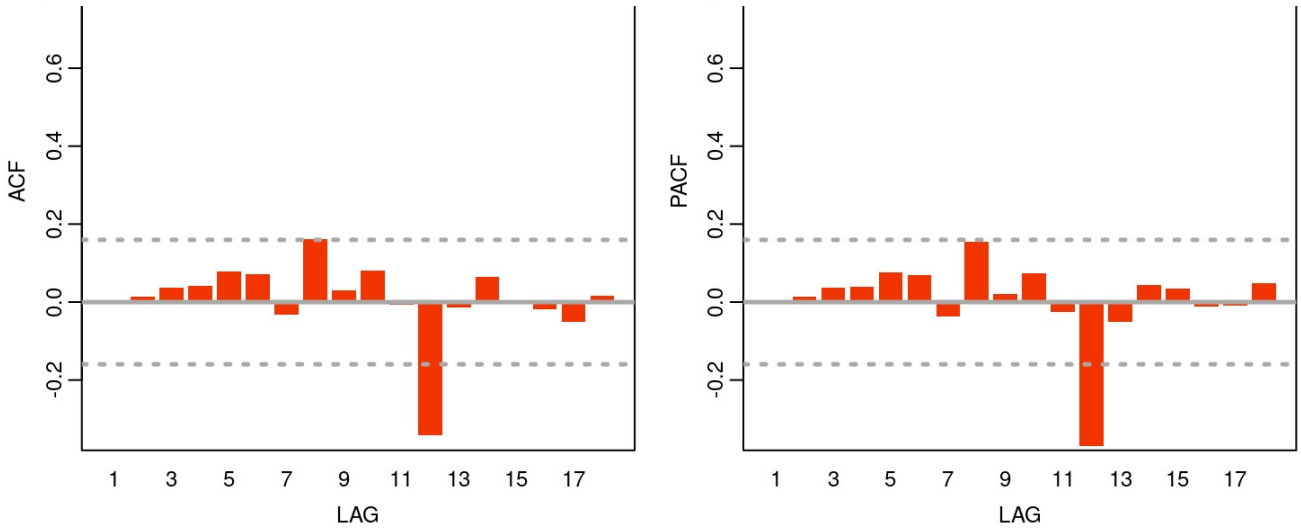
More or less. Thereafter, we look at the residuals for the model to determine if there is any serial correlation.

```
par(mfrow=c(1, 1))
arma$residuals %>%
  plot()
```



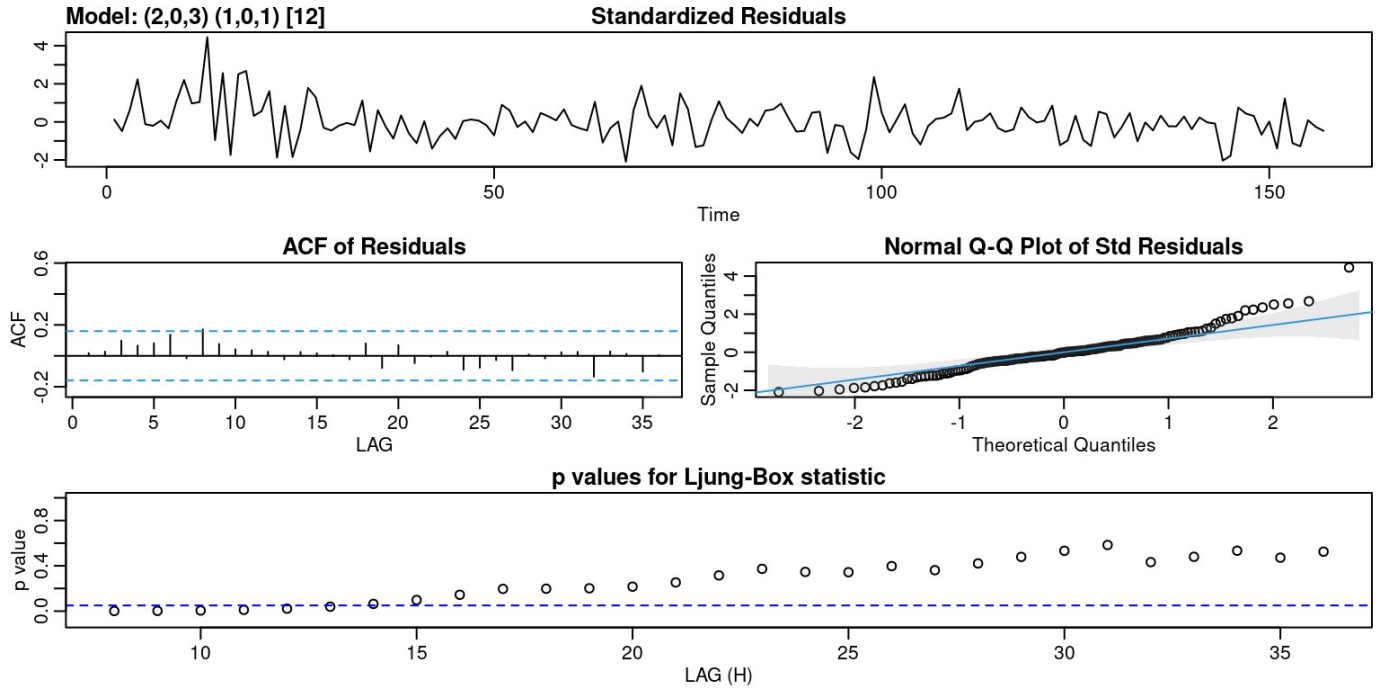

```
arma$residuals %>%
  ac()
```





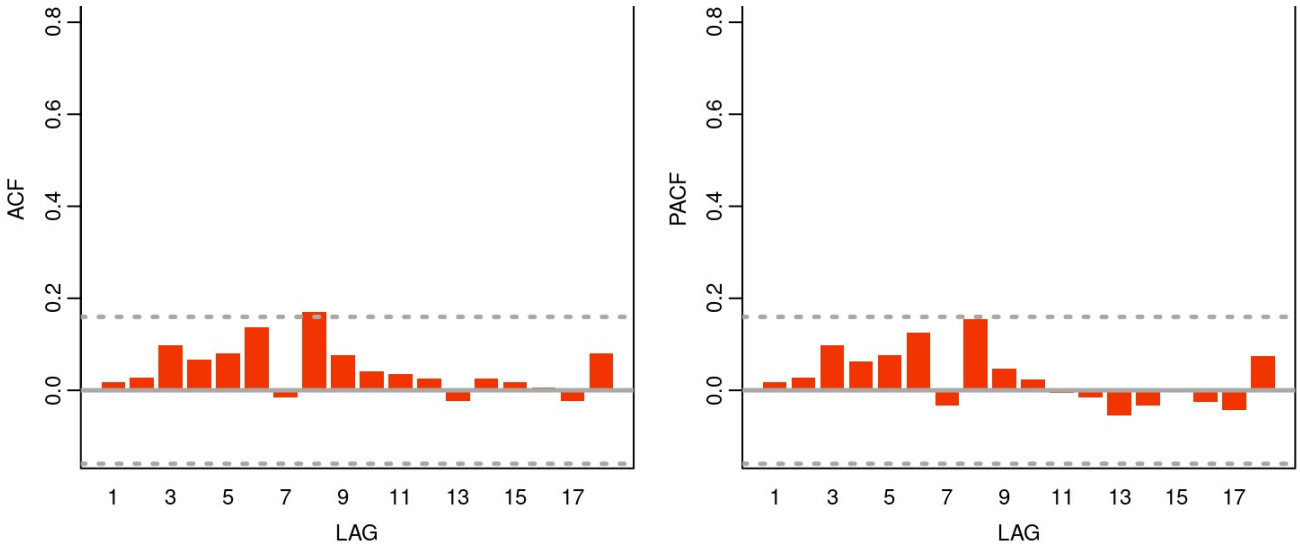
These results suggests that there is still some autocorrelation year-on-year. To estimate a arma model with seasonal components, we need to install the **astsa** package. In this case, we will fit the same model with an autoregressive and moving average seasonal `sarima(y,2,0,3,1,0,1,12)` makes use of the syntax `sarima(x,p,d,q,P,D,Q,S)`, which in this case will fit SARIMA(2,0,3)*(1,0,1)_{12}.

```
sarima_fit <- astsa::sarima(y,2,0,3,1,0,1,12) # sarima(x,p,d,q,P,D,Q,S) will fit SARIMA(2,0,
        3)*(1,0,1)_{12}
```



Thereafter, we can look at the residuals:

```
ac(as.numeric(sarima_fit$fit$residuals))
```



In this case we have effectively captured by the monthly seasonal component.

Bai, Jushan, and Pierre Perron. 2003. “Computation and Analysis of Multiple Structural Change Models.” *Journal of Applied Econometrics* 18 (1): 1–22.