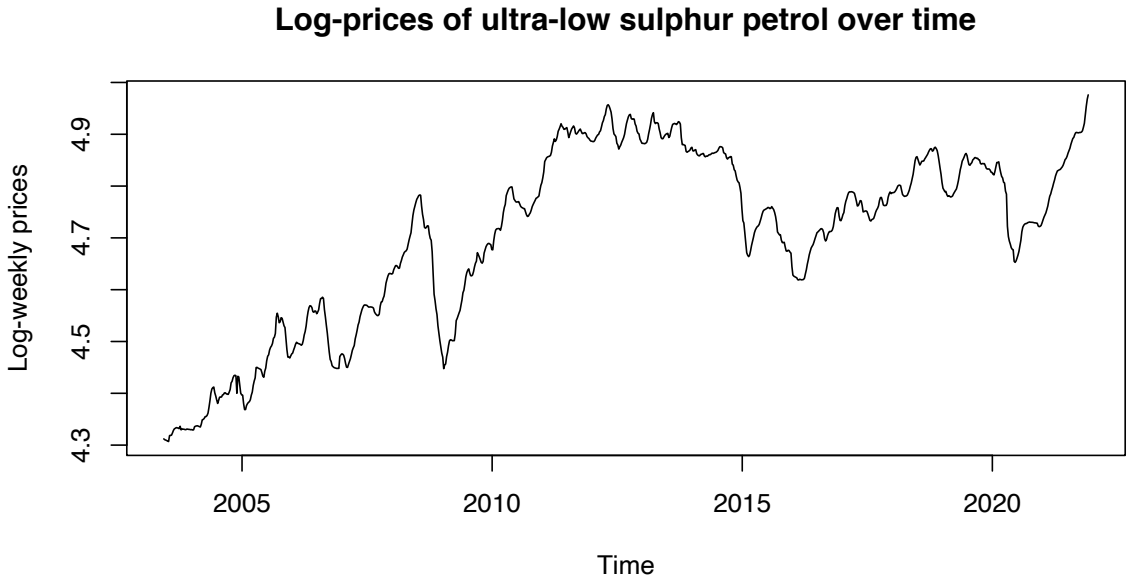
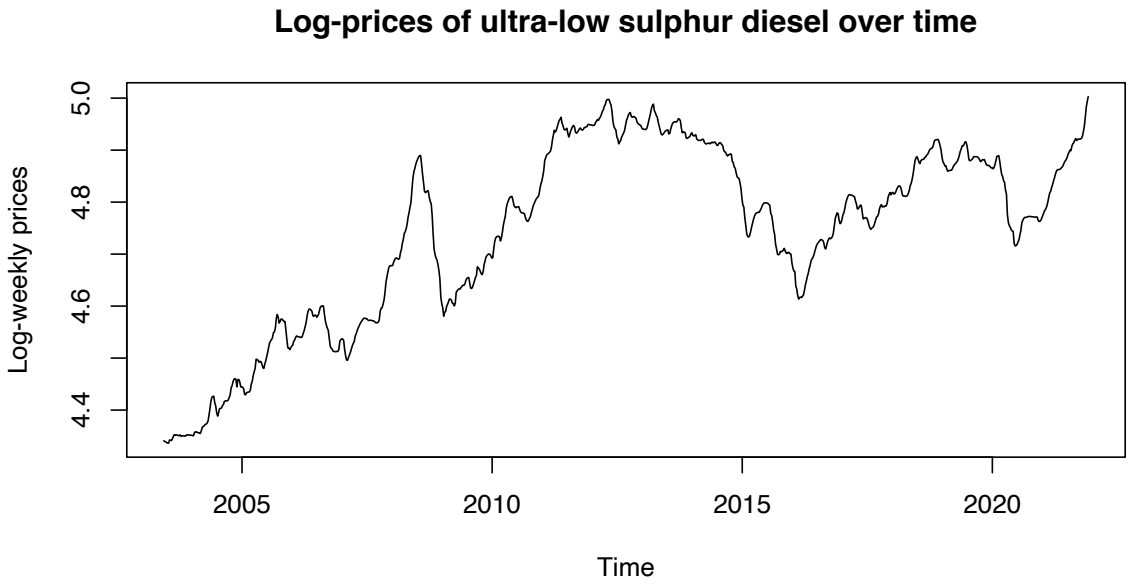
ST422 Project

Candidate number: 38388

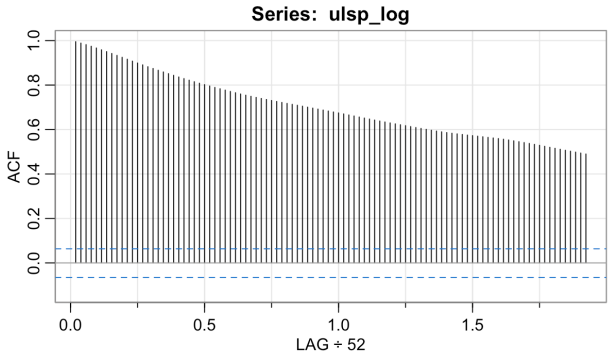
Question 1

The two time series given to our analysis are the weekly prices of fuel for ultra-low sulphur unleaded petrol (USLP) and ultra-low sulphur diesel (USLD). Both series consist of 962 weekly observations taken from 2003 to 2021. Since the data consists of prices, it might be helpful to consider the time series on a logarithmic scale. Indeed, the log-transformation helps to stabilize the variance of the data.
We first start by plotting the two transformed time series in order to visualise the whole data and look for any particular trends.

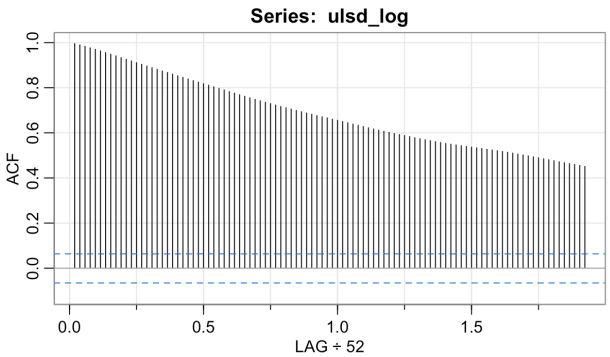




We can clearly see that both time series are not stationary and present an upward trend. We can verify that by plotting their sample auto-correlation functions.



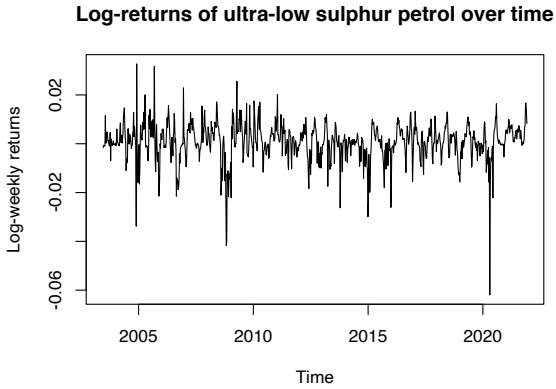
(a) Sample ACF of logged ULSP time series



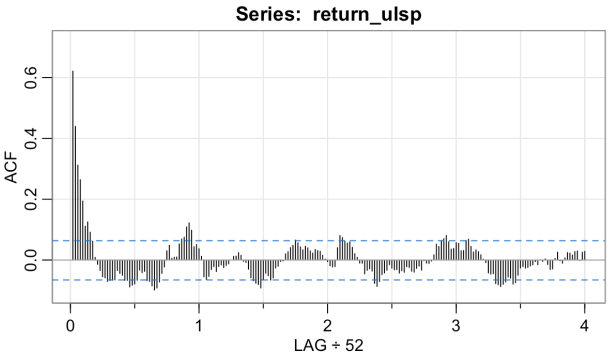
(b) Sample ACF of logged ULSD time series

From the ACF plots, we notice that almost all lags are outside the confidence bounds represented by the blue dashed lines. It confirms the existence of a strong trend that can be removed by taking the first order difference of the both time series. By doing so, we will obtain the time series log-returns which are good approximation of returns especially when the price changes are small.

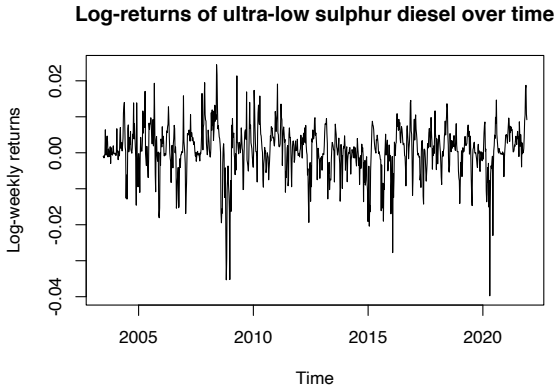
Let's plot the time series log-returns and their associated sample ACF functions in order to check whether the new time series look stationary.



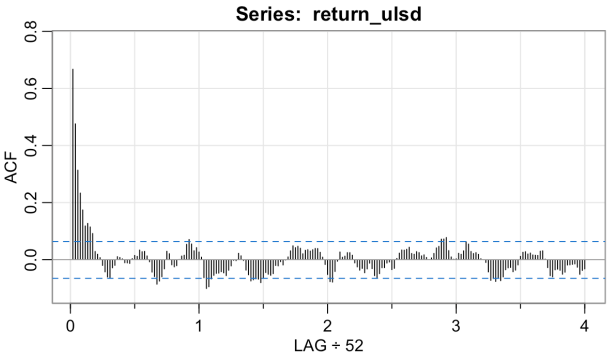
(a) Plot of log-returns of ULSP time series



(b) Sample ACF of ULSP log-returns time series



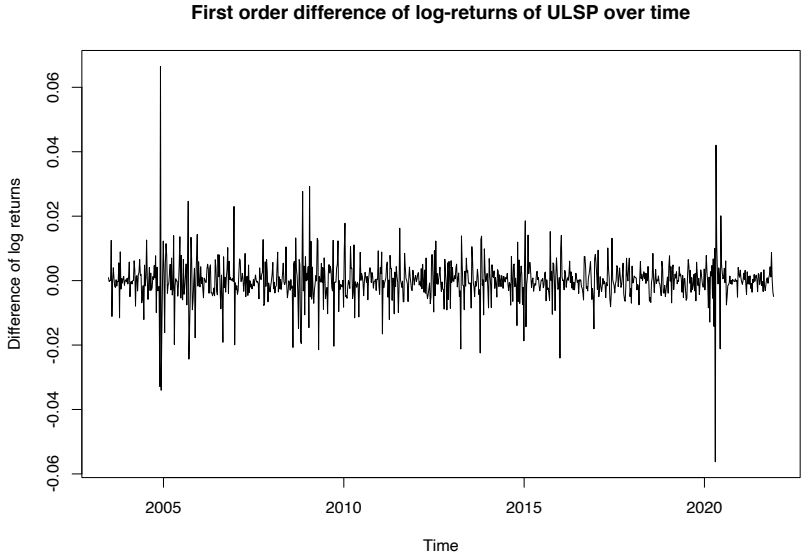
(c) Plot of log-returns of ULSD time series



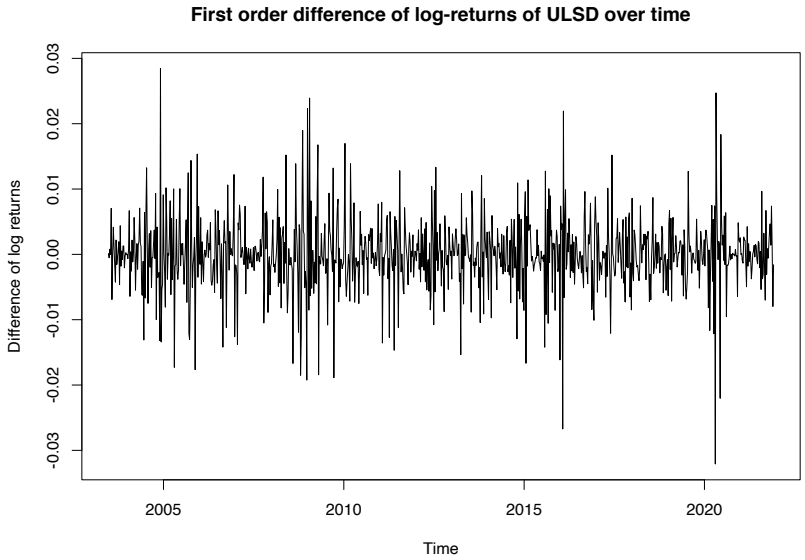
(d) Sample ACF of ULSD log-returns time series

In the above time series plots (a) and (c), we can see that the two time series look much more stationary when taking the first order difference. However, it seems that some trends remain for both series: a slight downward trend between 2010 and 2015 and a slight upward trend between 2015 and 2020. Moreover, we notice two important fluctuation periods around 2009 and 2020 which might respectively correspond to the sub-prime and Covid-19 crisis. When looking at the sample ACF plots, we can see that the first lags stick out for approximately ten weeks (0.2). In addition, for petrol log-returns, lags at 26, 36, 47, 78, 109, 120, 150 and 176 week slightly exceed the confidence bound, as well as, for lags at 36, 47, 54, 104, 150 and 176 week for diesel log-returns. Those lags suggest that the time series might contain multiple seasonality components that might cause those higher lags at different time points.

In order to ensure that both time series are stationary, and in, an effort to reduce the effect of the seasonal components, we can try to take the second order difference of the logged time series. By doing so we obtain the following graphs.



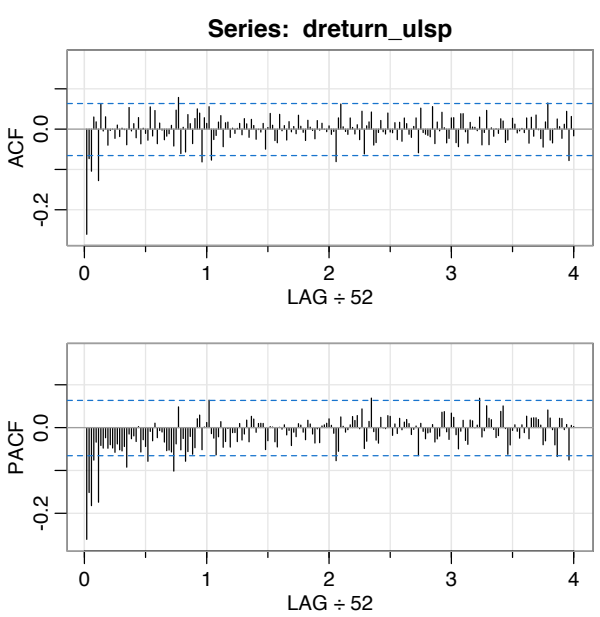
(a) Plot of difference of log-returns of ULSP time series



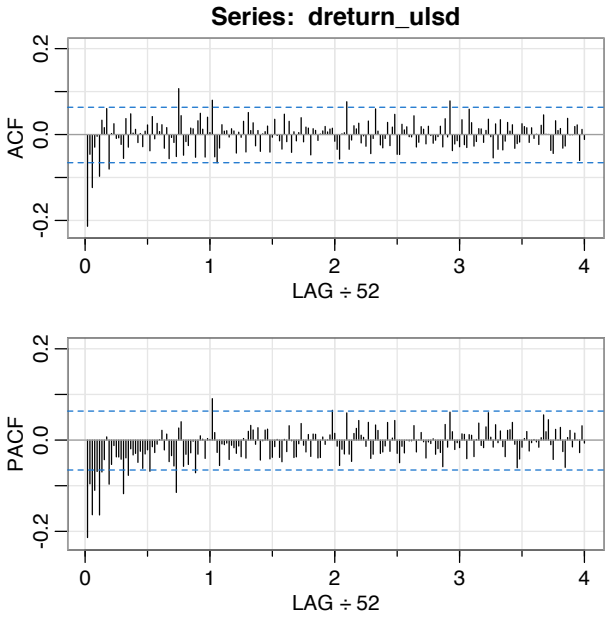
(b) Plot of difference of log-returns of ULSD time series

After taking the first order difference on the log-returns, both time series look stationary with no visible trend when looking at figure (a) and (b). However, the log-returns of ULSD still displays some important fluctuations that couldn't be much reduced by differentiate its log-returns.

Let's look at the ACF and PACF plots of both time series using the $acf2()$ command.



(a) Sample ACF and PACF plots of difference of log-returns for ULSP time series



(b) Sample ACF and PACF plots of difference of log-returns for ULSD time series

By looking at the ACF and PACF plots for the two time series, we can notice that the high first lags remained, as well as, the lags located on higher weeks. Therefore, the second order difference of log-prices didn't significantly contribute in reducing the size of the high lags.

Hence, the two logged-time series analysed presented some linear trend that we tried to reduce by taking the first order difference, which correspond to taking the log-returns of the ULSP and ULSD. The second order difference didn't improve the time series. Therefore, it might be more suitable to directly use log-returns for future modelling and forecasting.
If P_t^P and P_t^D denotes the price at time t for respectively ultra-low sulphur unleaded petrol and ultra-low sulphur unleaded diesel, then the log-returns are respectively:

$$r_t^P = \log(P_t^P / P_{t-1}^P)$$

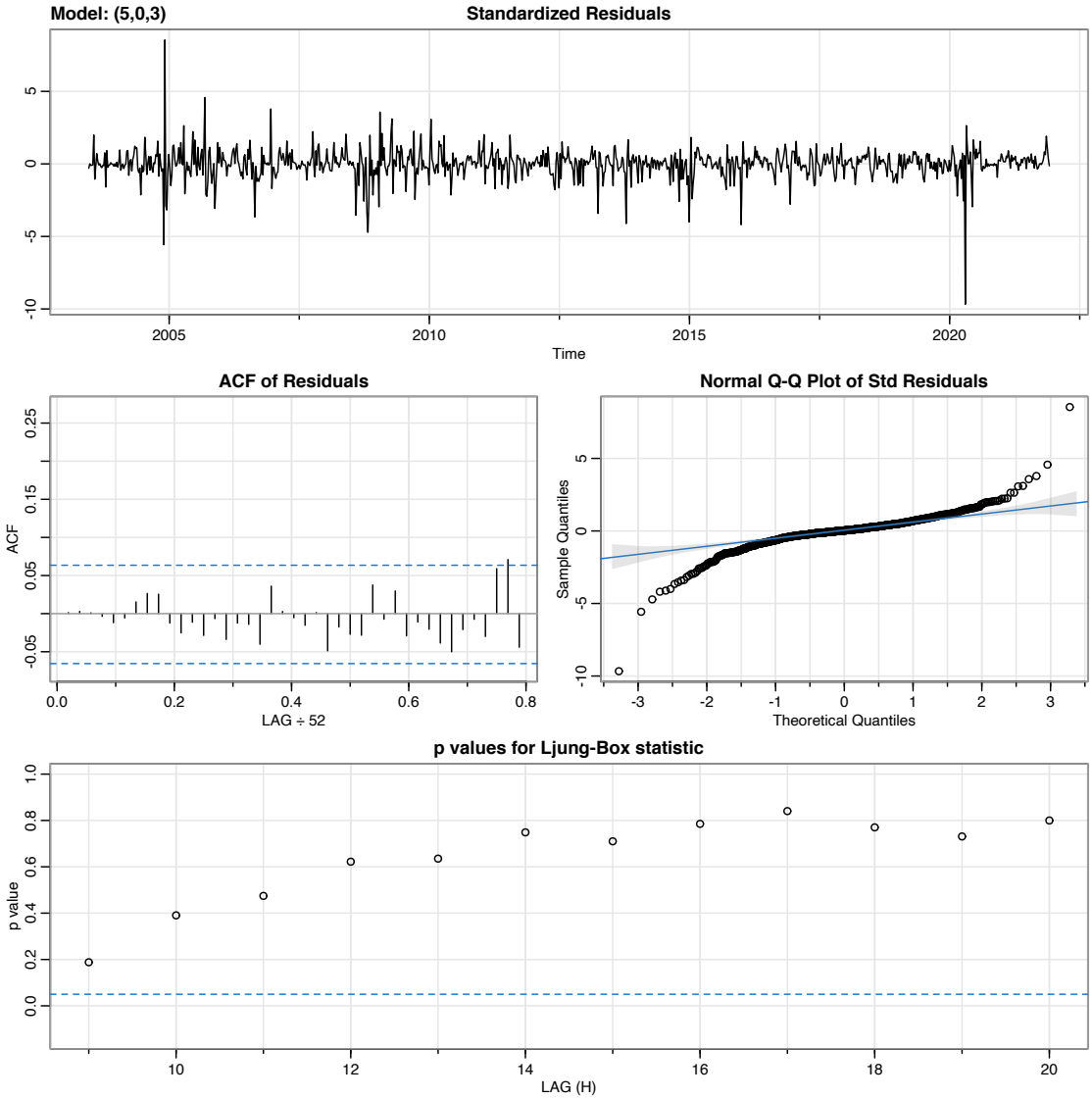
$$r_t^D = \log(P_t^D / P_{t-1}^D)$$

Question 2

To model time series, we first have to extract the trend and seasonal components, in order to model the microscopic part left. The log-returns of both ULSP and ULSD derived in question 1 seems to contain no visible trend, and therefore we will directly model the log-returns.
Our goal is to find a model with the lowest possible AIC that presents standardized residuals around -3 and 3 and Ljung-Box-Pierce test p-values greater than 0.05 for each lag. A good starting point is to use the *auto.arima*() function that will automatically test several models and outputs the "best" possible one.

Model for ULSP log-returns

The *auto.arima*() function for the ULSP log-returns outputs an ARMA(1,1) model with an AIC of -7401.69. Hence, we will start with this first model, and then modify the AR or MA order, in order to find a model with the lowest AIC.
By trial and error, we finally obtain an ARMA(5,3) model with an AIC of -7106.01. Let's have a look at its diagnostic plot by using the *sarima*() function which produces all diagnostic plots.



The standardized residual plot shows that the model fits better for some data points than for other data points. For example, we have extremely large residuals that are greater than 7 for the 2005 and 2020 period. This might suggests that the time series presents some change point in those periods and a rolling window approach might produce better fits. Moreover, despite the high lag at week 36, the ACF of residuals looks approximately like white noise which is good. Finally, all the p-values for the Ljung-Box static are greater than 0.05 which indicates that all the $\hat{\epsilon}$ are white noise and consequently, there is no significant correlation at certain lags.

In regards to our analysis in the question 1, we may try to fit a seasonal ARIMA model with period equal to 52 weeks (one year) and see whether it produces a model with an ever lower AIC. By fitting several models and selecting the one with the lowest AIC, we obtain a seasonal ARIMA(5,0,3)(0,0,1)[52] model with an AIC equal to -7104.56 which is greater than the AIC of our previous model. Therefore, we reject this seasonal model.

Hence, we believe that no model are satisfactory enough to fit the log-returns of ULSP without producing large residuals, especially for certain periods. Perhaps, a rolling window approach might produce better fits. However, the most satisfactory model found is an ARMA(5,3) model, because of its lowest AIC and its good diagnostic plots. The model's coefficients are:

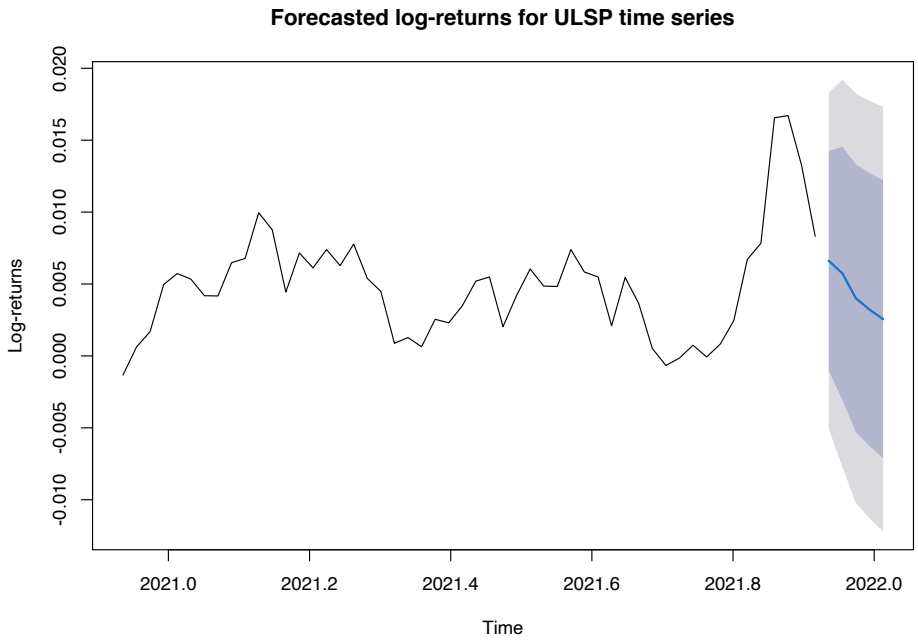
```
Series: return_ulsp
ARIMA(5,0,3) with non-zero mean

Coefficients:
          ar1      ar2     ar3     ar4     ar5     ma1     ma2     ma3   mean
      -0.8391  -0.1301  0.2134  0.3677  0.1507  1.4107  1.0072  0.4313  7e-04
s.e.   0.2095   0.1920  0.1255  0.1263  0.0338  0.2113  0.2903  0.1897  6e-04

sigma^2 estimated as 3.556e-05:  log likelihood=3563
AIC=-7106.01   AICc=-7105.77   BIC=-7057.33

Training set error measures:
                      ME        RMSE        MAE MPE MAPE      MASE        ACF1
Training set 4.806136e-06 0.005935193 0.00367983 NaN  Inf 0.4851576 0.001360947
```

Mathematically, the model can be written in the following way:

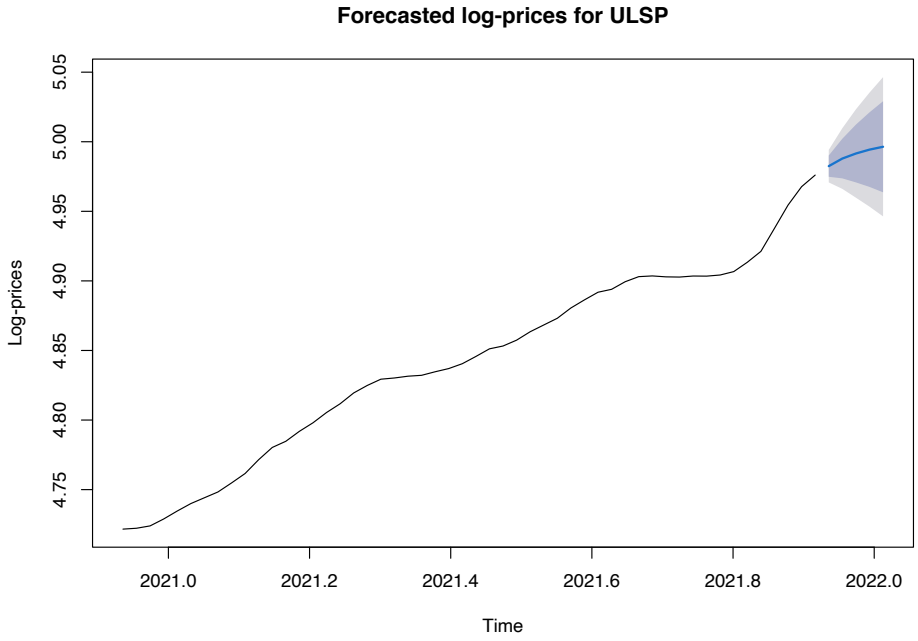
$$(1 + 0.83B + 0.13B^2 - 0.21B^3 - 0.37B^4 - 0.15B^5)r_t = (1 + 1.4B + B^2 + 0.4B^3)\epsilon_t$$

where $r_t = \log(P_t^P / P_{t-1}^P)$.

To produce forecast until the last week of December, we will use the $forecast()$ function. We obtain the following plotted predictions for the log-returns and the log-prices:



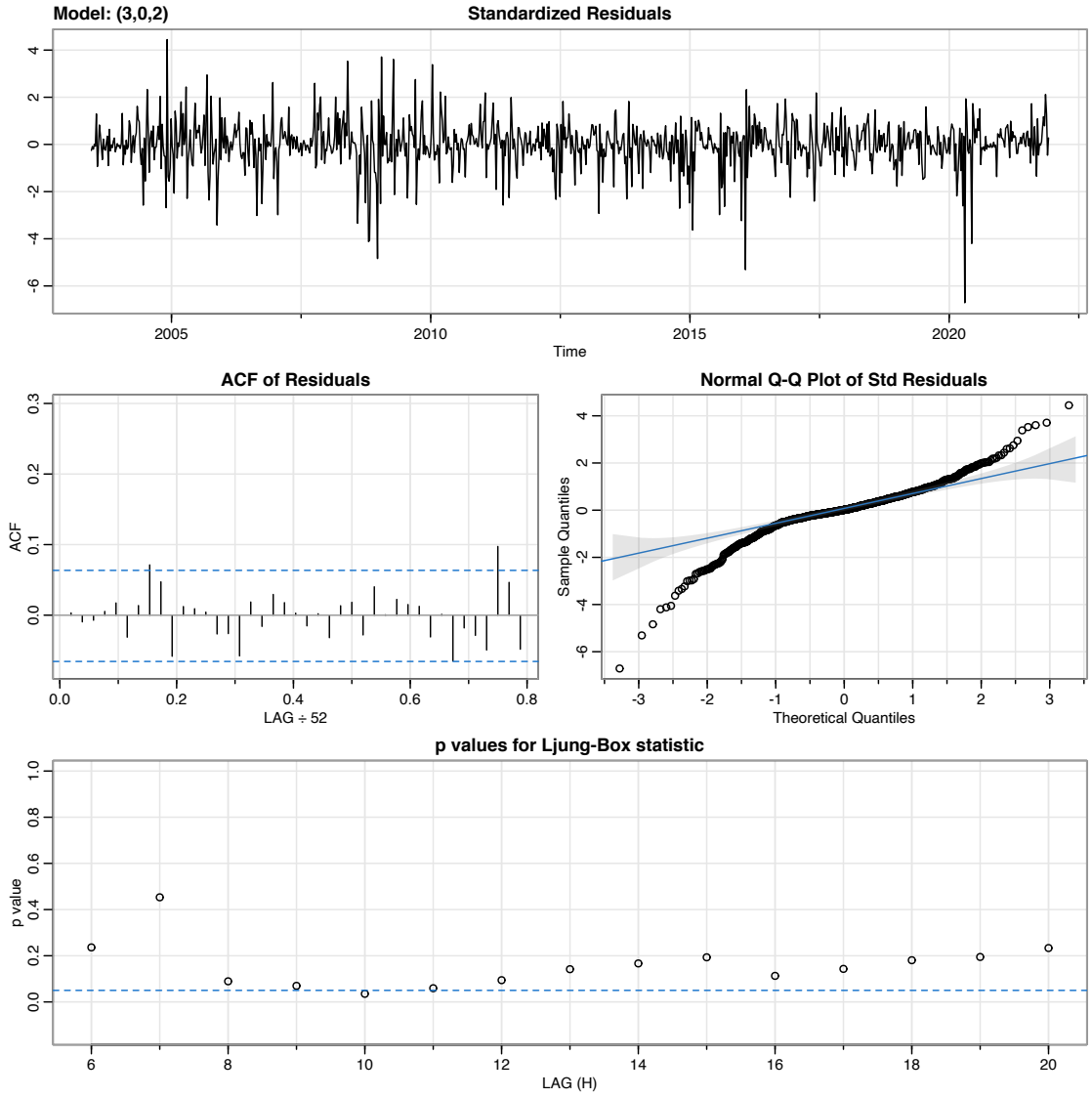
(a) Forecast of log-returns for ULSP until last week of December



(b) Forecast of log-prices for ULSP until last week of December

Model for ULSD log-returns

We will use the same process as for ULSP log-return time series in order to find an optimal model for ULSD log-return time series. The *auto.arima*() function outputs an ARMA(1,1) model with AIC equal to -7417.5. By trial and error, we found the lowest possible AIC equal to -7418.78 for an ARMA(3,2). Let's have a look at its diagnostic plots.



We can see that the standardized residuals present some import fluctuations with many standardized residuals greater than 4 or lower than -4. It suggests that the model doesn't fit well the data and a rolling window approach may produce better fits. In addition, we can clearly see in the bottom plot tha p-values from lag 8 to 12 are close or lower than 0.05, which implies that some $\hat{\epsilon}$ are not white noises and, therefore, time series presents certain correlation for certain lags. Therefore, the ARMA(3,2) is not a good fit for the log-return of ULSD time series.

Let's try an another model that would consider a seasonality of 52 periods. By trial and error, we obtain a seasonal ARIMA(3,0,2)(0,0,1)[52] model with an AIC equal to -7399.04 which is greater than the AIC of our previous model. Similarly, its diagnostic plots are no better, with several p-value for several lags below the threshold 0.05. Hence, we reject the mode.

Therefore, no models are satisfactory enough to fit the log-returns of ULSD without having some correlations at certain lags. The best model that we could find is an ARMA(3,2) model which has the smallest AIC among the other models we tried. The model's coefficients are:

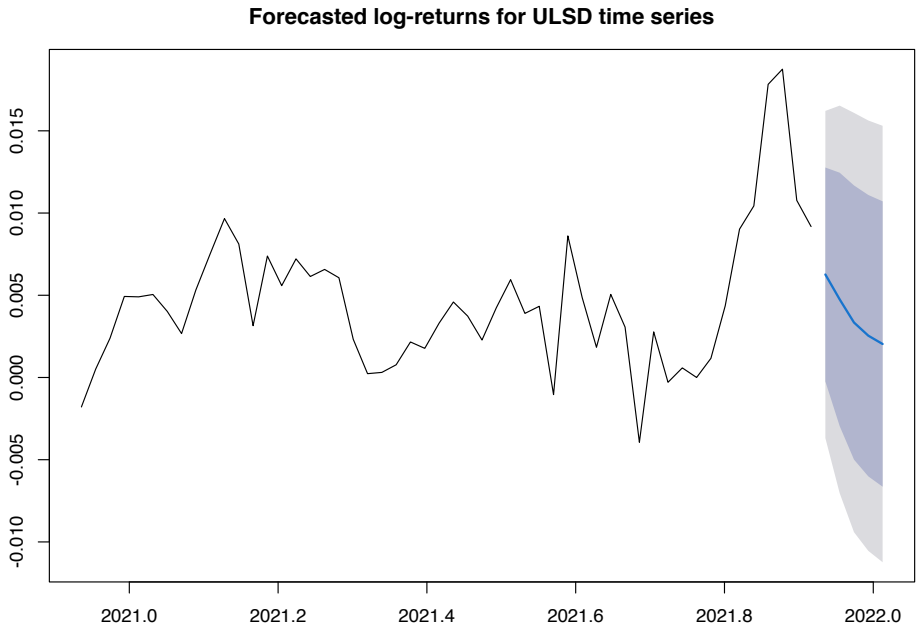
```
Series: return_ulsd
ARIMA(3,0,2) with non-zero mean

Coefficients:
          ar1     ar2     ar3     ma1     ma2   mean
      -0.8609  0.2442  0.5624  1.4959  0.7917  7e-04
s.e.   0.1164  0.1155  0.0752  0.1162  0.1336  5e-04

sigma^2 estimated as 2.576e-05:  log likelihood=3716.39
AIC=-7418.78   AICc=-7418.66   BIC=-7384.71

Training set error measures:
                       ME        RMSE         MAE MPE MAPE      MASE        ACF1
Training set 3.060399e-06 0.005059513 0.003386126 NaN  Inf 0.4815079 0.003228881
```

Mathematically, the model can be written in the following way:

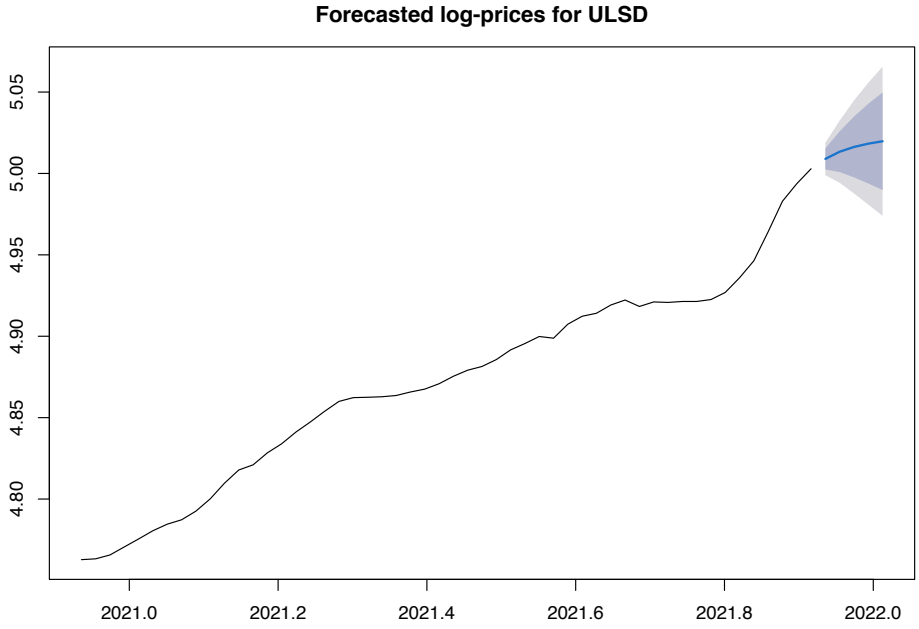
$$(1 + 0.86B - 0.24B^2 - 0.56B^3)r_t = (1 + 1.5B + 0.79B^2)\epsilon_t$$

where $r_t = \log(P_t^D / P_{t-1}^D)$.

We then derive the forecast for the log-prices of ULSD and the log-returns of ULSD until the last week of 2022. By plotting the forecasted point, we obtain the following plots:



(a) Forecast of log-returns for ULSD until last week of December



(b) Forecast of log-prices for ULSD until last week of December

Question 3

In the R-code in appendix 3, we first split the both data into training and validation data sets. Then, I created an R-function called *pred.roll*(). This function takes as argument some data, an integer D representing the window length and an integer A that moves the window forward A data points. For each window length, the function computes the next A predictions, and then derive the prediction errors for each window, to finally output the aggregate prediction errors. Therefore, if we have N windows for the data set, the function computes $A * (N - 1)$ predictions errors, since it can't compute the prediction error for the last window.

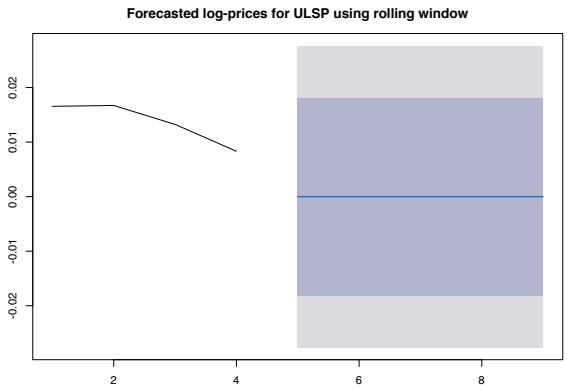
In addition, I created an another R-function called $automated_fit.roll()$ that takes as input some data and an integer rep which iterates the fit.roll function (rep-3) times, while comparing different window length and outputs the window length that has the lowest possible aggregate errors. We have to note that this function takes A = ceiling(D/3).

If we assume that the window length D is divisible by 3 and A = D/3, then the number of predictions performed by fit.roll is equal to:

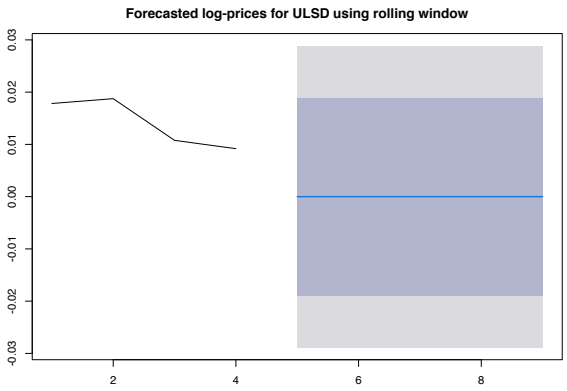
$$((n - D)/A) + 1 - 1) * A = n - D$$

where n is the data length. Indeed, the function starts predicting value from data point D onward. Hence, the sum of predictions of large window length will be biased since they will do less prediction, meaning less errors. Therefore, this bias can be ignored for small windows length, however, for window larger than 32 (8 months) for example, the bias will be significant. One way to solve the issue would be to add the model residuals for the first window to the aggregate prediction term.

With this in mind, we can start running the $automated_fit.roll()$ function on the ULSP and ULSD log-returns. We will take rep=32 for the 4 data sets, meaning that the function will test window length from 4 to 32 and output the optimal window length that produces the lowest aggregate error.
Despite the advantage for high window length, we obtain a similar window length of six weeks for both the training and validation data of the weekly log-returns of ULSP. Similarly, for the log-return of the ULSD data, we obtain a window length of six weeks. Hence, the rolling window matches for the training and validation data sets.
This result implies that the the model with the most accurate predictions will only take into account the six previous weeks in order to predict the next two coming weeks.
To predict the measurement until the last of December 2021, we have to make 5 prediction points. However, our window length can only predict for the next 2 week points. One simple way would be to use the past 5 week points to predict the next 5 week points. The prediction accuracy will decrease however, we do not expect significant fluctuations in the 5 weeks. By doing so, we obtain the following predictions:



(a) Forecast of log-returns for ULSP until last week of December



(b) Forecast of log-returns using for ULSD until last week of December

We can see that the predictions using a small window estimated that no changes will happen within five weeks, since all predictions are null. The predictions for log-returns in question 2 are all between 0.2 and 0.6% which is pretty close to the predictions made using rolling window equal to 0%. Hence, it seems that there is no significant differences between both predictions.

Question 4

By using the *mstl*() function provided in the forecast package in R, we can decompose the two time series into trend, seasonal and residual components. Let's plot the decomposed time series.

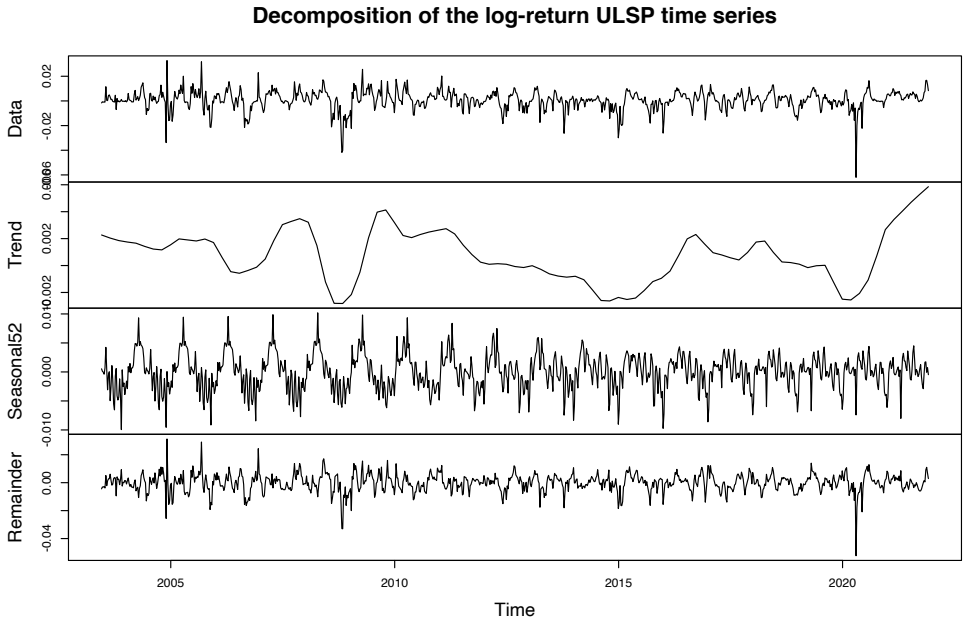


Figure 8: Decomposition of the ULSP log-return time series

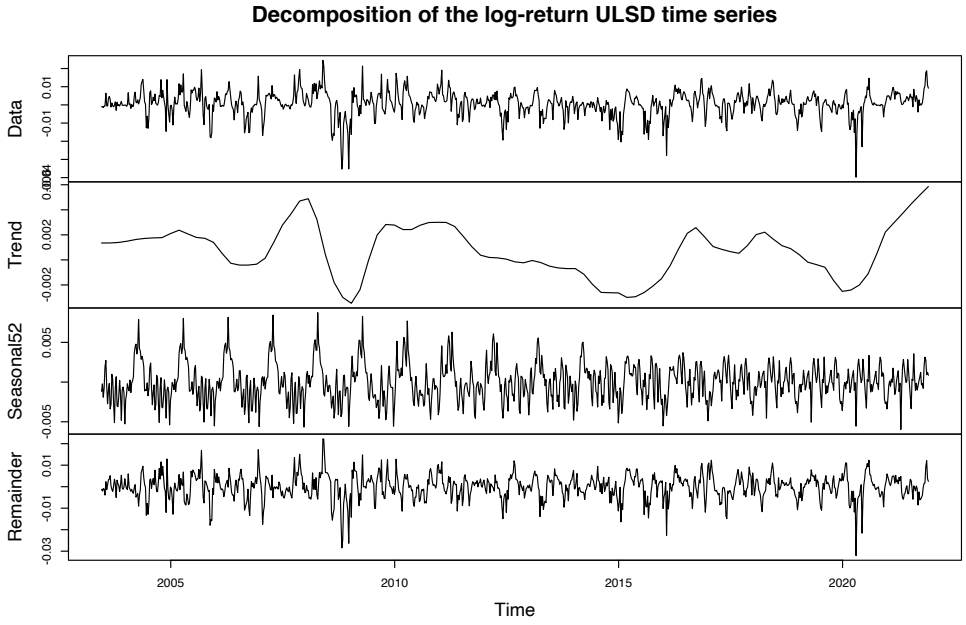


Figure 9: Decomposition of the ULSD log-return time series

After several researches, I finally found a written R function that can perform forecast on an object of type

mstl. This function is called $forecast.mstl()$ and the reference to the function can be find in the appendix. Hence, by using this function, we obtain the following predictions for ULSP and ULSD log-returns:

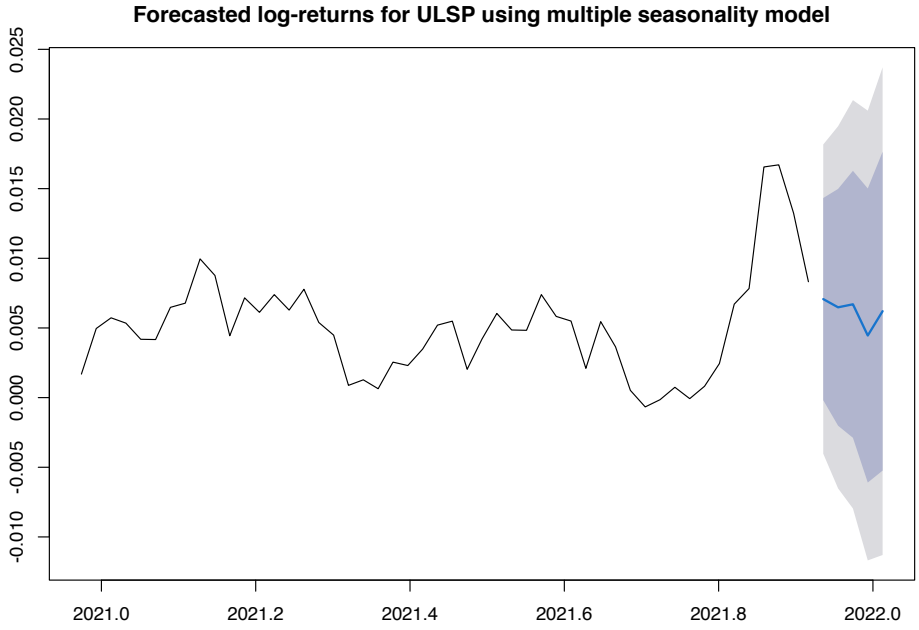


Figure 10: Prediction of the ULSP log-return using multiple seasonal model

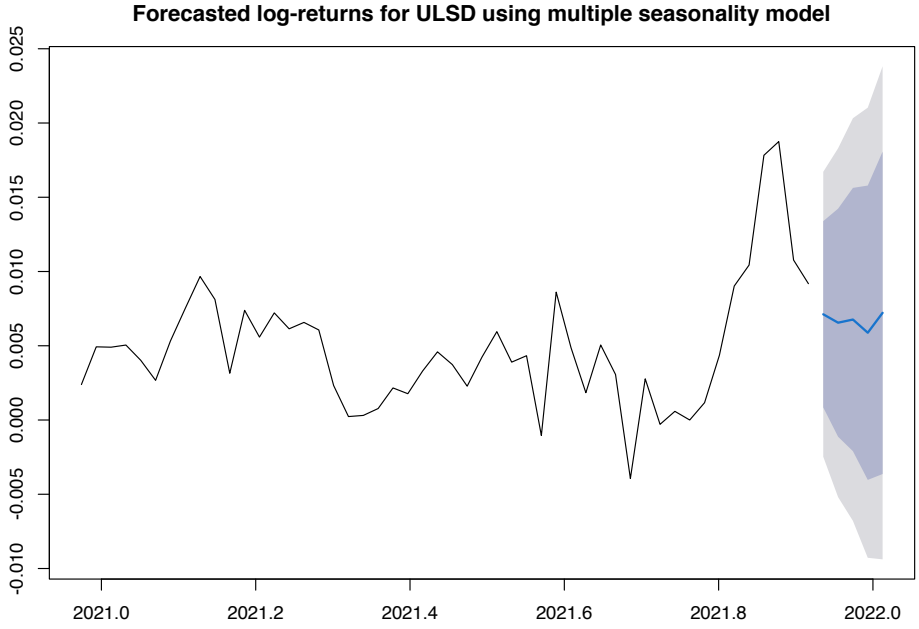


Figure 11: Prediction of the ULSD log-return using multiple seasonal model

All predictions are relatively close. However, those made using the mtls model are closer to the predictions made using the model built in question 2 compared to the model built using rolling window approach, except for the fifth predictions when they significantly differ.

Appendix

Link for forecasting mstl object

: https://rdrr.io/cran/forecast/src/R/mstl.Rsym-forecast.mstl

Code for question 1

:

```
library(forecast)
library(lubridate)
library(tseries)
library(zoo)
library(astsa)

fuel <- read.csv("Fuel_price.csv")

ulsp <- msts(fuel[,2], seasonal=365.25/7,
             start = decimal_date(as.Date("2003-06-09")))
ulsd <- msts(fuel[,3], seasonal=365.25/7,
             start = decimal_date(as.Date("2003-06-09")))

## Question 1
# Logarithmic transformation on data
ulsp_log <- log(ulsp)
ulsd_log <- log(ulsd)

# Plotting time series and its associated ACF
plot(ulsp_log, main= "Log-prices of ultra-low sulphur petrol over time",
xlab ="Time", ylab="Log-weekly prices" )
plot(ulsd_log, main= "Log-prices of ultra-low sulphur diesel over time",
xlab ="Time", ylab="Log-weekly prices" )

acf2(ulsp_log, max.lag=100)
acf2(ulsd_log, max.lag=100)

# Taking log returns of data
return_ulsp <- diff(log(ulsp))
return_ulsd <- diff(log(ulsd))

# Plotting time series and its associated ACF
plot(return_ulsp, main= "Log-returns of ultra-low sulphur petrol over time",
xlab ="Time", ylab="Log-weekly returns" )
acf2(return_ulsp)

plot(return_ulsd, main= "Log-returns of ultra-low sulphur diesel over time",
xlab ="Time", ylab="Log-weekly returns" )
acf2(return_ulsd)

# First order difference
dreturn_ulsp <- diff(return_ulsp)
dreturn_ulsd <- diff(return_ulsd)

# Plotting
```

```
plot(dreturn_ulsp, main= "First order difference of log-returns of ULSP over time",
xlab ="Time", ylab="Difference of log returns" )
acf2(dreturn_ulsp)

plot(dreturn_ulsd, main= "First order difference of log-returns of ULSD over time",
xlab ="Time", ylab="Difference of log returns" )
acf2(dreturn_ulsd)
```

Code for question 2

:

```
## Question 2
# Model for ULSP
# Start with auto.arima
auto_ulsp <- auto.arima(return_ulsp)
summary(auto_ulsp) # ARMA(1,1), AIC: -7101.69
tsdiag(auto_ulsp) # last point don't pass the test

# Best model with lowest AIC
mod_uslp <- Arima(return_ulsp, order=c(5,0,3)) # AIC -7106.01
summary(mod_uslp)
sarima(return_ulsp, 5,0,3)

# Try to add seasonality
mod_ulsp_season <- Arima(return_ulsp, order=c(5,0,3), seasonal = list(order=c(0,0,1),
period = 52))
summary(mod_ulsp_season) # ARIMA(5,0,3)(0,0,1)[52], AIC: -7104.56
sarima(return_ulsp, 5,0,3,0,0,1,52)

## Predictions using mod_ulsp
# Log-return predictions
pred_ulsp_return <- forecast(mod_uslp, 5)
plot(pred_ulsp_return, 52, main="Forecasted log-returns for ULSP time series",
xlab="Time", ylab="Log-returns")

# Log-prices predictions
pred_ulsp_price <- forecast(Arima(ulsp_log, order=c(5,1,3)), 5)
plot(pred_ulsp_price, 52, main="Forecasted log-prices for ULSP", xlab="Time",
ylab="Log-prices")

# Model for ULSD
# Start with auto.arima
auto_ulsd <- auto.arima(return_ulsd) # limit the search to 10 models because of
lengthy running time
summary(auto_ulsd) # suggests ARMA(1,0,1) with AIC: -7417.5
sarima(return_ulsd, 1,0,1)

# Best model with lowest AIC
mod_ulsd <- Arima(return_ulsd, order=c(3,0,2))
summary(mod_ulsd) # ARMA(3,2) with AIC: -7418.78
sarima(return_ulsd, 3,0,2)

# Try to add seasonality
```

```
mod_ulsd_s <- Arima(dreturn_ulsd, order=c(3,0,2), seasonal = list(order=c(0,0,1),
period = 52))
summary(mod_ulsd_s) # ARIMA(3,0,2)(0,0,1)[52], AIC: -7399.04
sarima(return_ulsd, 3,0,2,0,0,1,52)


## Predictions using mod_ulsd
# Log-return predictions
pred_ulsd_return <- forecast(mod_ulsd, 5)
plot(pred_ulsd_return, 52, main="Forecasted log-returns for ULSD time series",
xlab="Time", ylab="Log-returns")

# Log-prices predictions
pred_ulsd_price <- forecast(Arima(ulsd_log, order=c(3,1,2)), 5)
plot(pred_ulsd_price, 52, main="Forecasted log-prices for ULSD", xlab="Time",
ylab="Log-prices")
```

Code for question 3

:

```
## Question 3
# Splitting the data into training and validation data
n <- length(fuel[,2])
training_length <- ceiling(n*0.60)

ulsp_training <- msts(fuel[,2][1:training_length], seasonal=365.25/7,
                      start = decimal_date(as.Date("2003-06-09")))

ulsp_validation <- msts(fuel[,2][(training_length+1):n], seasonal=365.25/7,
                        start = decimal_date(as.Date("2003-06-09")))

ulsd_training <- msts(fuel[,3][1:training_length], seasonal=365.25/7,
                      start = decimal_date(as.Date("2003-06-09")))

ulsd_validation <- msts(fuel[,3][(training_length+1):n], seasonal=365.25/7,
                        start = decimal_date(as.Date("2003-06-09")))

return_ulsp_training <- diff(log(ulsp_training))
return_ulsp_valid <- diff(log(ulsp_validation))
return_ulsd_training <- diff(log(ulsd_training))
return_ulsd_valid <- diff(log(ulsd_validation))

## Rolling window functions

pred.roll <- function(data, D, A) {
  # Function that predicts the next A data points, using a window of length D each time
  # moving the window forward A data points. The fitted model for prediction is derived
  # from auto.arima(). Computes the prediction error for each window and outputs the final
  # aggregate error term, as well as, a list of fitted model.
  n <- length(data)
  N <- ceiling((n-D)/A) + 1
  fit <- list()
  pred_error <- 0
```

```
  for (i in 1:N){
    data.ts <- msts(data[((i-1)*A+1):(min((i-1)*A+D,n))], seasonal=365.25/7 )
    fit[[i]] = auto.arima(data.ts)
    #tsdiag(fit[[i]])
    pred <- forecast(fit[[i]], A)
    if ((min((i-1)*A+D,n)) + A < n) {
      error <- sum((data[((min((i-1)*A+D,n))+1):((min((i-1)*A+D,n)) + A)]
      - pred$mean)^2)
      # predict next A data points
      pred_error <- pred_error + as.numeric(error) # aggregate error component
    }
  }
  return(list(fit, pred_error))
}


automated_pred.roll <- function(data, rep) {
  # Function that iterates the fit.roll (rep-3) times and outputs the window length
  # with lowest prediction error. Take A = ceiling(D/3)
  min_error <- 1000 # some large integer that will never be exceeded
  D_min <- 0
  for (D in 4:rep) {
    A <- ceiling(D/3)
    temp <- pred.roll(data, D, A)
    if (temp[[2]] < min_error) {
      min_error <- temp[[2]]
      D_min <- D
    }
  }
  return(list(D_min, min_error))
}

# Optimal window length for ULSP
automated_pred.roll(return_ulsp_training, 32) #  D = 6, with error = 0.03391066
automated_pred.roll(return_ulsp_valid, 32) # D = 6, with error = 0.01729922
# Same window length

# Optimal window length for ULSD
automated_pred.roll(return_ulsd_training, 32) # D = 6, with error = 0.02670696
automated_pred.roll(return_ulsd_valid, 32) # D = 6, with error = 0.01426493
# Same window length



## Forecasting the next 5 data points
## Predictions
ulsp_wind_fcst <- forecast(auto.arima(return_ulsp[(n-4):n]), 5)
plot(ulsp_wind_fcst, main="Forecasted log-prices for ULSP using rolling window",
xlab="Time", ylab="Log-prices")
ulsd_wind_fcst <- forecast(auto.arima(return_ulsd[(n-4):n]), 5)
plot(ulsd_wind_fcst, main="Forecasted log-prices for ULSD using rolling window",
xlab="Time", ylab="Log-prices")

# Prediction points from question 2
```

```
pred_ulsp_return$mean
pred_ulsd_return$mean
```

Code for question 4

:

```
## Question 4
# Perform decomposition of the ULSP and ULSD log-returns time series
ulsp_decomp <- mstl(return_ulsp, iterate = 100)
ulsd_decomp <- mstl(return_ulsd, iterate = 100)

# Plot each component
plot(ulsp_decomp, main="Decomposition of the log-return ULSP time series")
plot(ulsd_decomp, main="Decomposition of the log-return ULSD time series")

# We can forecast the data using the stlf() function
ulsp_decomp_fcst <- forecast.mstl(ulsp_decomp, h=5)
plot(ulsp_decomp_fcst, main="Forecasted log-returns for ULSP using
multiple seasonality model", xlab="Time", ylab="Log-prices", include = 50)

ulsd_decomp_fcst <- forecast.mstl(ulsd_decomp, h=5)
plot(ulsd_decomp_fcst, main="Forecasted log-returns for ULSD using
multiple seasonality model", xlab="Time", ylab="Log-prices", include = 50)

# Compare to previous prediction point
ulsp_decomp_fcst
pred_ulsp_return$mean

ulsd_decomp_fcst
pred_ulsd_return$mean
```

Used code for the multiple seasonal predictions

:

```
#' Multiple seasonal decomposition
#'
#' Decompose a time series into seasonal, trend and remainder components.
#' Seasonal components are estimated iteratively using STL. Multiple seasonal periods are
#' allowed. The trend component is computed for the last iteration of STL.
#' Non-seasonal time series are decomposed into trend and remainder only.
#' In this case, \code{\link[stats]{supsmu}} is used to estimate the trend.
#' Optionally, the time series may be Box-Cox transformed before decomposition.
#' Unlike \code{\link[stats]{stl}}, \code{mstl} is completely automated.
#' @param x Univariate time series of class \code{msts} or \code{ts}.
#' @param iterate Number of iterations to use to refine the seasonal component.
#' @param s.window Seasonal windows to be used in the  decompositions. If scalar,
#' the same value is used for all seasonal components. Otherwise, it should be a vector
#' of the same length as the number of seasonal components (or longer).
#' @param ... Other arguments are passed to \code{\link[stats]{stl}}.
#' @inheritParams forecast
#'
#' @seealso \code{\link[stats]{stl}}, \code{\link[stats]{supsmu}}
#' @examples
#' library(ggplot2)
#' mstl(taylor) %>% autoplot()
#' mstl(AirPassengers, lambda = "auto") %>% autoplot()
```

```r
#' @export
mstl <- function(x, lambda = NULL, iterate = 2, s.window = 7+4*seq(6), ...) {
  # What is x?
  origx <- x
  n <- length(x)
  if ("msts" %in% class(x)) {
    msts <- attributes(x)$msts
    if (any(msts >= n / 2)) {
      warning("Dropping seasonal components with fewer than two full periods.")
      msts <- msts[msts < n / 2]
      x <- forecast::msts(x, seasonal.periods = msts)
    }
    msts <- sort(msts, decreasing = FALSE)
  }
  else if ("ts" %in% class(x)) {
    msts <- frequency(x)
    iterate <- 1L
  }
  else {
    x <- as.ts(x)
    msts <- 1L
  }
  # Check dimension
  if (!is.null(dim(x))) {
    if (NCOL(x) == 1L) {
      x <- x[, 1]
    }
  }

  # Replace missing values if necessary
  if (anyNA(x)) {
    x <- na.interp(x, lambda = lambda)
  }

  # Transform if necessary
  if (!is.null(lambda)) {
    x <- forecast::BoxCox(x, lambda = lambda)
    lambda <- attr(x, "lambda")
  }
  tt <- seq_len(n)

  # Now fit stl models with only one type of seasonality at a time
  if (msts[1L] > 1) {
    seas <- as.list(rep(0, length(msts)))
    deseas <- x
    if (length(s.window) == 1L) {
      s.window <- rep(s.window, length(msts))
    }
    iterate <- pmax(1L, iterate)
    for (j in seq_len(iterate))
    {
      for (i in seq_along(msts))
      {
        deseas <- deseas + seas[[i]]
```

```
        fit <- stl(ts(deseas, frequency = msts[i]), s.window = s.window[i], ...)
        seas[[i]] <- msts(seasonal(fit), seasonal.periods = msts)
        attributes(seas[[i]]) <- attributes(x)
        deseas <- deseas - seas[[i]]
      }
    }
    trend <- msts(trendcycle(fit), seasonal.periods = msts)
  }
  else {
    msts <- NULL
    deseas <- x
    trend <- ts(stats::supsmu(seq_len(n), x)$y)
  }
  attributes(trend) <- attributes(x)

  # Put back NAs
  deseas[is.na(origx)] <- NA

  # Estimate remainder
  remainder <- deseas - trend

  # Package into matrix
  output <- cbind(origx, trend)
  if (!is.null(msts)) {
    for (i in seq_along(msts)) {
      output <- cbind(output, seas[[i]])
    }
  }
  output <- cbind(output, remainder)
  colnames(output)[1L:2L] <- c("Data", "Trend")
  if (!is.null(msts)) {
    colnames(output)[2L + seq_along(msts)] <- paste0("Seasonal", round(msts, 2))
  }
  colnames(output)[NCOL(output)] <- "Remainder"

  if (length(msts) > 1) {
    attr(output, "seasonal.periods") <- msts
    return(structure(output,
                     seasonal.periods = msts,
                     class = c("mstl", "mts", "msts", "ts")
    ))
  }

  return(structure(output, class = c("mstl", "mts", "ts")))
}

#' @rdname autoplot.seas
#' @export
autoplot.mstl <- function(object, ...) {
  autoplot.mts(object, facets = TRUE, ylab = "", ...)
}

#' Forecasting using stl objects
#'
```

```
#' Forecasts of STL objects are obtained by applying a non-seasonal forecasting
#' method to the seasonally adjusted data and re-seasonalizing using the last
#' year of the seasonal component.
#'
#' \code{stlm} takes a time series \code{y}, applies an STL decomposition, and
#' models the seasonally adjusted data using the model passed as
#' \code{modelfunction} or specified using \code{method}. It returns an object
#' that includes the original STL decomposition and a time series model fitted
#' to the seasonally adjusted data. This object can be passed to the
#' \code{forecast.stlm} for forecasting.
#'
#' \code{forecast.stlm} forecasts the seasonally adjusted data, then
#' re-seasonalizes the results by adding back the last year of the estimated
#' seasonal component.
#'
#' \code{stlf} combines \code{stlm} and \code{forecast.stlm}. It takes a
#' \code{ts} argument, applies an STL decomposition, models the seasonally
#' adjusted data, reseasonalizes, and returns the forecasts. However, it allows
#' more general forecasting methods to be specified via
#' \code{forecastfunction}.
#'
#' \code{forecast.stl} is similar to \code{stlf} except that it takes the STL
#' decomposition as the first argument, instead of the time series.
#'
#' Note that the prediction intervals ignore the uncertainty associated with
#' the seasonal component. They are computed using the prediction intervals
#' from the seasonally adjusted series, which are then reseasonalized using the
#' last year of the seasonal component. The uncertainty in the seasonal
#' component is ignored.
#'
#' The time series model for the seasonally adjusted data can be specified in
#' \code{stlm} using either \code{method} or \code{modelfunction}. The
#' \code{method} argument provides a shorthand way of specifying
#' \code{modelfunction} for a few special cases. More generally,
#' \code{modelfunction} can be any function with first argument a \code{ts}
#' object, that returns an object that can be passed to \code{\link{forecast}}.
#' For example, \code{forecastfunction=ar} uses the \code{\link{ar}} function
#' for modelling the seasonally adjusted series.
#'
#' The forecasting method for the seasonally adjusted data can be specified in
#' \code{stlf} and \code{forecast.stl} using either \code{method} or
#' \code{forecastfunction}. The \code{method} argument provides a shorthand way
#' of specifying \code{forecastfunction} for a few special cases. More
#' generally, \code{forecastfunction} can be any function with first argument a
#' \code{ts} object, and other \code{h} and \code{level}, which returns an
#' object of class \code{\link{forecast}}. For example,
#' \code{forecastfunction=thetaf} uses the \code{\link{thetaf}} function for
#' forecasting the seasonally adjusted series.
#'
#' @param y A univariate numeric time series of class \code{ts}
#' @param object An object of class \code{stl} or \code{stlm}. Usually the
#' result of a call to \code{\link[stats]{stl}} or \code{stlm}.
#' @param method Method to use for forecasting the seasonally adjusted series.
#' @param modelfunction An alternative way of specifying the function for
```

```
#' modelling the seasonally adjusted series. If \code{modelfunction} is not
#' \code{NULL}, then \code{method} is ignored. Otherwise \code{method} is used
#' to specify the time series model to be used.
#' @param model Output from a previous call to \code{stlm}. If a \code{stlm}
#' model is passed, this same model is fitted to y without re-estimating any
#' parameters.
#' @param forecastfunction An alternative way of specifying the function for
#' forecasting the seasonally adjusted series. If \code{forecastfunction} is
#' not \code{NULL}, then \code{method} is ignored. Otherwise \code{method} is
#' used to specify the forecasting method to be used.
#' @param etsmodel The ets model specification passed to
#' \code{\link[forecast]{ets}}. By default it allows any non-seasonal model. If
#' \code{method!="ets"}, this argument is ignored.
#' @param xreg Historical regressors to be used in
#' \code{\link[forecast]{auto.arima}()} when \code{method=="arima"}.
#' @param newxreg Future regressors to be used in
#' \code{\link[forecast]{forecast.Arima}()}.
#' @param h Number of periods for forecasting.
#' @param level Confidence level for prediction intervals.
#' @param fan If \code{TRUE}, level is set to seq(51,99,by=3). This is suitable
#' for fan plots.
#' @param s.window Either the character string ``periodic'' or the span (in
#' lags) of the loess window for seasonal extraction.
#' @param t.window A number to control the smoothness of the trend. See
#' \code{\link[stats]{stl}} for details.
#' @param robust If \code{TRUE}, robust fitting will used in the loess
#' procedure within \code{\link[stats]{stl}}.
#' @param allow.multiplicative.trend If TRUE, then ETS models with
#' multiplicative trends are allowed. Otherwise, only additive or no trend ETS
#' models are permitted.
#' @param x Deprecated. Included for backwards compatibility.
#' @param ... Other arguments passed to \code{forecast.stl},
#' \code{modelfunction} or \code{forecastfunction}.
#' @inheritParams forecast
#'
#' @return \code{stlm} returns an object of class \code{stlm}. The other
#' functions return objects of class \code{forecast}.
#'
#' There are many methods for working with \code{\link{forecast}} objects
#' including \code{summary} to obtain and print a summary of the results, while
#' \code{plot} produces a plot of the forecasts and prediction intervals. The
#' generic accessor functions \code{fitted.values} and \code{residuals} extract
#' useful features.
#' @author Rob J Hyndman
#' @seealso \code{\link[stats]{stl}}, \code{\link{forecast.ets}},
#' \code{\link{forecast.Arima}}.
#' @keywords ts
#' @examples
#'
#' tsmod <- stlm(USAccDeaths, modelfunction = ar)
#' plot(forecast(tsmod, h = 36))
#'
#' decomp <- stl(USAccDeaths, s.window = "periodic")
#' plot(forecast(decomp))
```

```
#' @export
forecast.stl <- function(object, method = c("ets", "arima", "naive", "rwdrift"), etsmodel =
                         forecastfunction = NULL,
                         h = frequency(object$time.series) * 2, level = c(80, 95), fan = FA
                         lambda = NULL, biasadj = NULL, xreg = NULL, newxreg = NULL, allow
  method <- match.arg(method)
  if (is.null(forecastfunction)) {
    if (method != "arima" && (!is.null(xreg) || !is.null(newxreg))) {
      stop("xreg and newxreg arguments can only be used with ARIMA models")
    }
    if (method == "ets") {
      # Ensure non-seasonal model
      if (substr(etsmodel, 3, 3) != "N") {
        warning("The ETS model must be non-seasonal. I'm ignoring the seasonal component s
        substr(etsmodel, 3, 3) <- "N"
      }
      forecastfunction <- function(x, h, level, ...) {
        fit <- ets(na.interp(x), model = etsmodel, allow.multiplicative.trend = allow.mult
        return(forecast(fit, h = h, level = level))
      }
    }
    else if (method == "arima") {
      forecastfunction <- function(x, h, level, ...) {
        fit <- auto.arima(x, xreg = xreg, seasonal = FALSE, ...)
        return(forecast(fit, h = h, level = level, xreg = newxreg))
      }
    }
    else if (method == "naive") {
      forecastfunction <- function(x, h, level, ...) {
        rwf(x, drift = FALSE, h = h, level = level, ...)
      }
    }
    else if (method == "rwdrift") {
      forecastfunction <- function(x, h, level, ...) {
        rwf(x, drift = TRUE, h = h, level = level, ...)
      }
    }
  }
  if (is.null(xreg) != is.null(newxreg)) {
    stop("xreg and newxreg arguments must both be supplied")
  }
  if (!is.null(newxreg)) {
    if (NROW(as.matrix(newxreg)) != h) {
      stop("newxreg should have the same number of rows as the forecast horizon h")
    }
  }
  if (fan) {
    level <- seq(51, 99, by = 3)
  }

  if ("mstl" %in% class(object)) {
    seasonal.periods <- attr(object, "seasonal.periods")
    if (is.null(seasonal.periods)) {
      seasonal.periods <- frequency(object)
```

```
  }
  seascomp <- matrix(0, ncol = length(seasonal.periods), nrow = h)
  for (i in seq_along(seasonal.periods))
  {
    mp <- round(seasonal.periods[i], 2)
    n <- NROW(object)
    colname <- paste0("Seasonal", mp)
    seascomp[, i] <- rep(object[n - rev(seq_len(mp)) + 1, colname], trunc(1 + (h - 1) / r
  }
  lastseas <- rowSums(seascomp)
  xdata <- object[, "Data"]
  seascols <- grep("Seasonal", colnames(object))
  allseas <- rowSumsTS(object[, seascols, drop = FALSE])
  series <- NULL
}
else if ("stl" %in% class(object)) {
  m <- frequency(object$time.series)
  n <- NROW(object$time.series)
  lastseas <- rep(seasonal(object)[n - (m:1) + 1], trunc(1 + (h - 1) / m))[1:h]
  xdata <- ts(rowSums(object$time.series))
  tsp(xdata) <- tsp(object$time.series)
  allseas <- seasonal(object)
  series <- deparse(object$call$x)
}
else {
  stop("Unknown object class")
}

# De-seasonalize
x.sa <- seasadj(object)
# Forecast
fcast <- forecastfunction(x.sa, h = h, level = level, ...)

# Reseasonalize
fcast$mean <- fcast$mean + lastseas
fcast$upper <- fcast$upper + lastseas
fcast$lower <- fcast$lower + lastseas
fcast$x <- xdata
fcast$method <- paste("STL + ", fcast$method)
fcast$series <- series
# fcast$seasonal <- ts(lastseas[1:m],frequency=m,start=tsp(object$time.series)[2]-1+1/m)
fcast$fitted <- fitted(fcast) + allseas
fcast$residuals <- fcast$x - fcast$fitted

if (!is.null(lambda)) {
  fcast$x <- InvBoxCox(fcast$x, lambda)
  fcast$fitted <- InvBoxCox(fcast$fitted, lambda)
  fcast$mean <- InvBoxCox(fcast$mean, lambda, biasadj, fcast)
  fcast$lower <- InvBoxCox(fcast$lower, lambda)
  fcast$upper <- InvBoxCox(fcast$upper, lambda)
  attr(lambda, "biasadj") <- biasadj
  fcast$lambda <- lambda
}
```

```
  return(fcast)
}

#' @export
forecast.mstl <- function(object, method = c("ets", "arima", "naive", "rwdrift"), etsmodel
                          forecastfunction = NULL,
                          h = frequency(object) * 2, level = c(80, 95), fan = FALSE,
                          lambda = NULL, biasadj = NULL, xreg = NULL, newxreg = NULL, allow
  forecast.stl(
    object,
    method = method, etsmodel = etsmodel,
    forecastfunction = forecastfunction, h = h, level = level, fan = fan, lambda = lambda,
    biasadj = biasadj, xreg = xreg, newxreg = newxreg, allow.multiplicative.trend = allow.
  )
}


# rowSums for mts objects
#
# Applies rowSums and returns ts with same tsp attributes as input. This
# allows the result to be added to other time series with different lengths
# but overlapping time indexes.
# param mts a matrix or multivariate time series
# return a vector of rowsums which is a ts if the \code{mts} is a ts
rowSumsTS <- function (mts) {
  the_tsp <- tsp(mts)
  ret <- rowSums(mts)
  if (is.null(the_tsp)){
    ret
  } else {
    tsp(ret) <- the_tsp
    as.ts(ret)
  }
}

# Function takes time series, does STL decomposition, and fits a model to seasonally adjus
# But it does not forecast. Instead, the result can be passed to forecast().
#' @rdname forecast.stl
#' @export
stlm <- function(y, s.window = 7+4*seq(6), robust = FALSE, method = c("ets", "arima"), mod
                 etsmodel = "ZZN", lambda = NULL, biasadj = FALSE, xreg = NULL, allow.mult
  method <- match.arg(method)

  # Check univariate
  if (NCOL(x) > 1L) {
    stop("y must be a univariate time series")
  } else {
    if (!is.null(ncol(x))) {
      if (ncol(x) == 1L) { # Probably redundant check
        x <- x[, 1L]
      }
    }
  }
  # Check x is a seasonal time series
```

```
tspx <- tsp(x)
if (is.null(tspx)) {
  stop("y is not a seasonal ts object")
} else if (tspx[3] <= 1L) {
  stop("y is not a seasonal ts object")
}

# Transform data if necessary
origx <- x
if (!is.null(lambda)) {
  x <- BoxCox(x, lambda)
  lambda <- attr(x, "lambda")
}

# Do STL decomposition
stld <- mstl(x, s.window = s.window, robust = robust)

if (!is.null(model)) {
  if (inherits(model$model, "ets")) {
    modelfunction <- function(x, ...) {
      return(ets(x, model = model$model, use.initial.values = TRUE, ...))
    }
  }
  else if (inherits(model$model, "Arima")) {
    modelfunction <- function(x, ...) {
      return(Arima(x, model = model$model, xreg = xreg, ...))
    }
  }
  else if (!is.null(model$modelfunction)) {
    if ("model" %in% names(formals(model$modelfunction))) {
      modelfunction <- function(x, ...) {
        return(model$modelfunction(x, model = model$model, ...))
      }
    }
  }
  if (is.null(modelfunction)) {
    stop("Unknown model type")
  }
}
# Construct modelfunction if not passed as an argument
else if (is.null(modelfunction)) {
  if (method != "arima" && !is.null(xreg)) {
    stop("xreg arguments can only be used with ARIMA models")
  }
  if (method == "ets") {
    # Ensure non-seasonal model
    if (substr(etsmodel, 3, 3) != "N") {
      warning("The ETS model must be non-seasonal. I'm ignoring the seasonal component s
      substr(etsmodel, 3, 3) <- "N"
    }
    modelfunction <- function(x, ...) {
      return(ets(
        x,
        model = etsmodel,
```

```
          allow.multiplicative.trend = allow.multiplicative.trend, ...
        ))
      }
    }
    else if (method == "arima") {
      modelfunction <- function(x, ...) {
        return(auto.arima(x, xreg = xreg, seasonal = FALSE, ...))
      }
    }
  }

  # De-seasonalize
  x.sa <- seasadj(stld)

  # Model seasonally adjusted data
  fit <- modelfunction(x.sa, ...)
  fit$x <- x.sa

  # Fitted values and residuals
  seascols <- grep("Seasonal", colnames(stld))
  allseas <- rowSumsTS(stld[, seascols, drop = FALSE])

  fits <- fitted(fit) + allseas
  res <- residuals(fit)
  if (!is.null(lambda)) {
    fits <- InvBoxCox(fits, lambda, biasadj, var(res))
    attr(lambda, "biasadj") <- biasadj
  }

  return(structure(list(
    stl = stld, model = fit, modelfunction = modelfunction, lambda = lambda,
    x = origx, series = deparse(substitute(y)), m = frequency(origx), fitted = fits, residu
  ), class = "stlm"))
}

#' @rdname forecast.stl
#' @export
forecast.stlm <- function(object, h = 2 * object$m, level = c(80, 95), fan = FALSE,
                          lambda = object$lambda, biasadj = NULL, newxreg = NULL, allow.mu
  if (!is.null(newxreg)) {
    if (nrow(as.matrix(newxreg)) != h) {
      stop("newxreg should have the same number of rows as the forecast horizon h")
    }
  }
  if (fan) {
    level <- seq(51, 99, by = 3)
  }

  # Forecast seasonally adjusted series
  if (is.element("Arima", class(object$model)) && !is.null(newxreg)) {
    fcast <- forecast(object$model, h = h, level = level, xreg = newxreg, ...)
  } else if (is.element("ets", class(object$model))) {
    fcast <- forecast(
      object$model,
```

```
      h = h, level = level,
      allow.multiplicative.trend = allow.multiplicative.trend, ...
    )
  } else {
    fcast <- forecast(object$model, h = h, level = level, ...)
  }

  # In-case forecast method uses different horizon length (such as using xregs)
  h <- NROW(fcast$mean)
  # Forecast seasonal series with seasonal naive
  seasonal.periods <- attributes(object$stl)$seasonal.periods
  if (is.null(seasonal.periods)) {
    seasonal.periods <- frequency(object$stl)
  }
  seascomp <- matrix(0, ncol = length(seasonal.periods), nrow = h)
  for (i in seq_along(seasonal.periods))
  {
    mp <- seasonal.periods[i]
    n <- NROW(object$stl)
    colname <- paste0("Seasonal", round(mp, 2))
    seascomp[, i] <- rep(object$stl[n - rev(seq_len(mp)) + 1, colname], trunc(1 + (h - 1) /
  }
  lastseas <- rowSums(seascomp)
  xdata <- object$stl[, "Data"]
  seascols <- grep("Seasonal", colnames(object$stl))
  allseas <- rowSumsTS(object$stl[, seascols, drop = FALSE])
  series <- NULL

  #  m <- frequency(object$stl$time.series)
  n <- NROW(xdata)

  # Reseasonalize
  fcast$mean <- fcast$mean + lastseas
  fcast$upper <- fcast$upper + lastseas
  fcast$lower <- fcast$lower + lastseas
  fcast$method <- paste("STL + ", fcast$method)
  fcast$series <- object$series
  # fcast$seasonal <- ts(lastseas[1:m],frequency=m,start=tsp(object$stl$time.series)[2]-1+
  # fcast$residuals <- residuals()
  fcast$fitted <- fitted(fcast) + allseas
  fcast$residuals <- residuals(fcast)

  if (!is.null(lambda)) {
    fcast$fitted <- InvBoxCox(fcast$fitted, lambda)
    fcast$mean <- InvBoxCox(fcast$mean, lambda, biasadj, fcast)
    fcast$lower <- InvBoxCox(fcast$lower, lambda)
    fcast$upper <- InvBoxCox(fcast$upper, lambda)
    attr(lambda, "biasadj") <- biasadj
    fcast$lambda <- lambda
  }
  fcast$x <- object$x

  return(fcast)
}
```

```
#' @rdname forecast.stl
#'
#' @examples
#'
#' plot(stlf(AirPassengers, lambda = 0))
#' @export
stlf <- function(y, h = frequency(x) * 2, s.window = 7+4*seq(6), t.window = NULL, robust =
  seriesname <- deparse(substitute(y))

  # Check univariate
  if (NCOL(x) > 1L) {
    stop("y must be a univariate time series")
  } else {
    if (!is.null(ncol(x))) {
      if (ncol(x) == 1L) { # Probably redundant check
        x <- x[, 1L]
      }
    }
  }
  # Check x is a seasonal time series
  tspx <- tsp(x)
  if (is.null(tspx)) {
    stop("y is not a seasonal ts object")
  } else if (tspx[3] <= 1L) {
    stop("y is not a seasonal ts object")
  }

  if (!is.null(lambda)) {
    x <- BoxCox(x, lambda)
    lambda <- attr(x, "lambda")
  }

  fit <- mstl(x, s.window = s.window, t.window = t.window, robust = robust)
  fcast <- forecast(fit, h = h, lambda = lambda, biasadj = biasadj, ...)

  # if (!is.null(lambda))
  # {
  #   fcast$x <- origx
  #   fcast$fitted <- InvBoxCox(fcast$fitted, lambda)
  #   fcast$mean <- InvBoxCox(fcast$mean, lambda)
  #   fcast$lower <- InvBoxCox(fcast$lower, lambda)
  #   fcast$upper <- InvBoxCox(fcast$upper, lambda)
  #   fcast$lambda <- lambda
  # }

  fcast$series <- seriesname

  return(fcast)
}

#' @rdname is.ets
#' @export
is.stlm <- function(x) {
```

```
  inherits(x, "stlm")
}
```