The Merit Order Effect in the German and Austrian Power Market

Seminar Paper

Felix Germaine Manuel Pfeuffer Bruno Puri

Berlin, August 12, 2018

We show for the combinend German and Austrian energy market that in the period of 2015–2017 an increase of 1 GWh in the hourly average of daily solar and wind energy production has, on average, a negative effect of 0.49 €/MWh and 1.20 €/MWh respectively on the energy price. In our analysis we follow a similar study by Clò, Cataldi, and Zoppoli for the Italian energy market. However, our results are not as robust as we use a smaller dataset, only encompassing the three years, and do not control for the gas price. Our source code can be accessed at https://github.com/mpff/spl2018-bfm/.

Summer Term 2018
Statistical Programming Languages
Ladislaus von Bortkiewicz Chair of Statistics
Humboldt-Universität zu Berlin

Contents

1. Introduction and Theory

In today's era of climate change, the need to reduce global greenhouse gas emissions by substituting traditional energy production methods by renewables is greater than ever. While such a transition is necessary to preserve our environment, it could also have some beneficial direct effects on energy prices. The latter effect, known as the "merit order effect", arises from the fact that the production of renewable electricity has very low marginal costs compared to other production processes. Therefore, a rise in the production of renewable electricity leads to a shift of the supply curve to the right, which should lead to a fall in electricity prices.

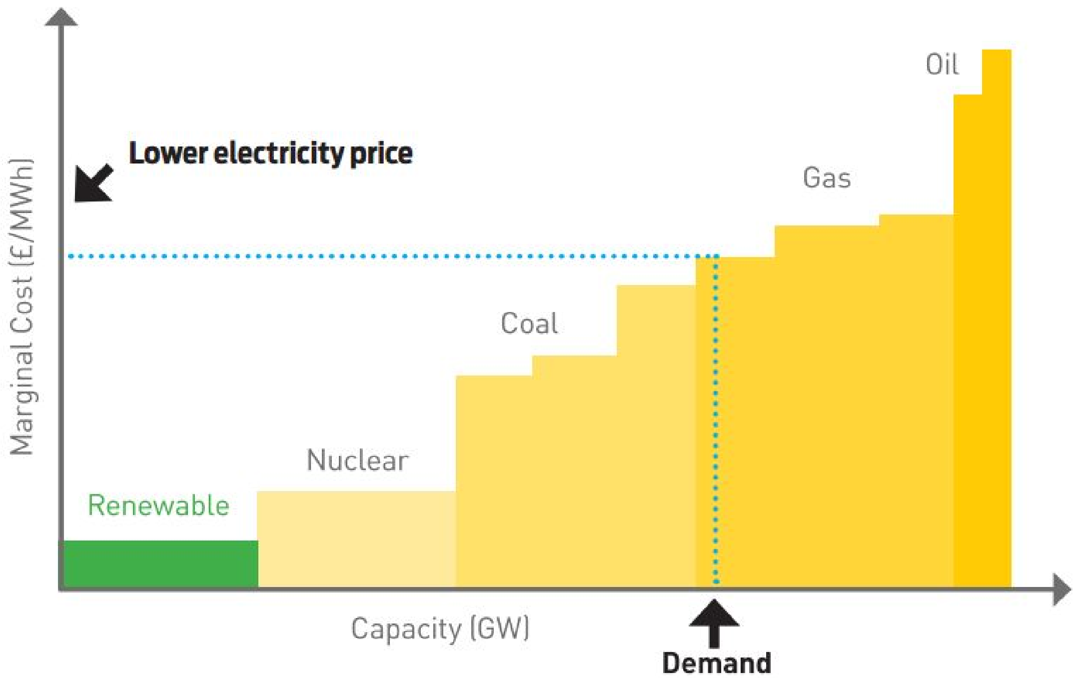


Figure 1: Illustration of the merit order effect. (Source: Greenpeace Unearthed)

The aim of our project is to quantify this effect for solar and wind generation in the coupled German and Austrian wholesale electricity market.

1.1. Model specification

We will follow the methodology of Clò, Cataldi, and Zoppoli, in their paper "The merit-order effect in the Italian power market: The impact of solar and wind generation on national wholesale electricity prices", where they regressed daily national wholesale

electricity prices on solar generation, wind generation, demand and a seasonal component (Dummy variables for Years, Months and Days of the week) in the italian market.
Unfortunately, as gas price data was not available for the German and Austrian market, we could not implement the final regression of the Italian paper. Nevertheless, their estimation of the impact of renewables on the electricity price does not differ much when gas prices are added to the specification.

$$Price_t = \beta_{DEM} * Demand_t + \beta_{SOL} * Solar_t + \beta_{WIND} * Wind_t + \beta_{Seasonal} * D_t + \epsilon_t \quad (1)$$

$Price_t$: Average wholesale electricity price (day ahead) [EUR/MWh]

$Demand_t$: Forecasted hourly averaged daily demand (day ahead) [GWh/h]

$Solar_t$: Forecasted hourly averaged daily solar generation (day ahead) [GWh/h]

$Wind_t$: Forecasted hourly averaged daily wind generation (day ahead) [GWh/h]

D_t : Dummy Matrix controlling for year, month and day of the week

Although hourly and quarter-hourly data could be found, the regression only takes into account aggregated daily data, as according to the authors, a higher resolution would lead to unwanted noise. Furthermore, we control for seasonal effects with help of a dummy variable matrix for years, months and days of the week, as our electricity data has strong seasonal patterns. As mentioned by Clò, Cataldi, and Zoppoli, the main explanatory variables of Equation (1) should be exogenous: Electrical demand is usually highly inelastic and does not react strongly to price changes. Wind and Solar generation mainly depend on meteorological factors, that certainly do not depend on electrical prices. Of course, as we are dealing with time series data, it is important to take care of possible auto-correlation of the error terms.

1.2. Regression and testing methodology

In a first step, we will check for trend-stationarity of the relevant data (price, demand, solar generation and wind generation) using the Augmented Dickey-Fuller test and the Philipps-Perron test (which is robust to autocorrelation and heteroscedasticity).
Then we will run the ordinary least squares (OLS) regression according to Equation (1)

in order to examine the auto-correlation structure of the error terms by plotting the ACF and the PACF and by performing the Durbin-Watson test.
Finally, as done by Clò, Cataldi, and Zoppoli, we will use the Prais-Winsten estimation method in order to model the error terms of Equation (1) through an AR(1) process. Again, the ACF and PACF will be checked and the Durbin-Watson test for autocorrelation will be performed in order to asses whether our estimation method is valid.

1.3. Data sources

Before going into a more detailed description of the process leading to our final regression, we shall present our data sources and formats below:

Data	Country	Source	Resolution	Unit
Price	DE-AT	ELSPOT	1 Hour	[EUR/MWh]
Demand	DE-AT-LU	ENTSOE	15 Minutes	[MW]
Solar, Wind	DE	Netztransparenz	15 Minutes	[MW]
Solar, Wind	AT	APG	15 Minutes	[MW]

DE: Germany , AT: Austria, LU: Luxemburg

2. Design and Code Structure

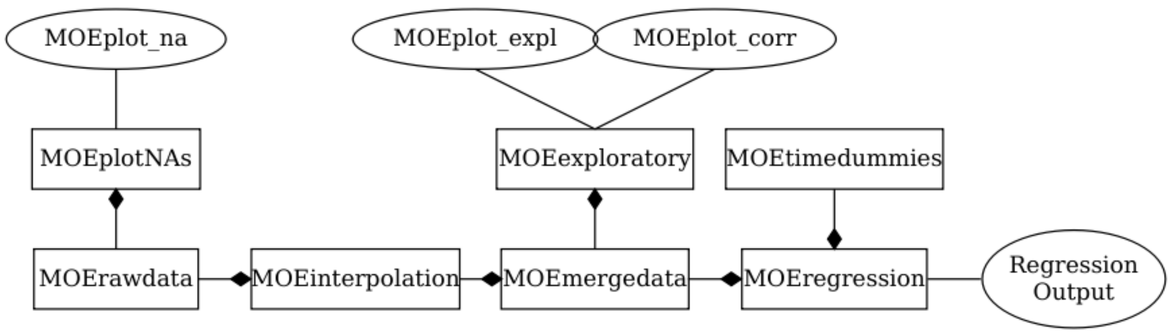
The goal of our project is to run a variety of tests and regressions on Austrian and German energy market data. From a programming perspective this is a rather easy task, as most tests and types of regression are already implemented in various `R` packages. A major part of our project was preparing the data for the application of these prepackaged functions. In short, the core of our work consists in these two steps:

Step 1: Prepare the raw data and produce a coherent datacube.

Step 2: Run the tests and regression.

Additionally a couple of plots were created to explore our data set and visualize some key features.
In the QuantNet design philosophy we split the codebase for these steps into multiple smaller segments, each built for a single purpose. These Quantlets are supposed to be run in order, some Quantlets creating data that is used by other Quantlets for further

processing, testing or plotting. Figure 2 gives an overview over this projects Quantlet structure. It was created using the `DiagrammeR` package.



MOEqletflow

Figure 2: This projects Quantlet dependency chart. Quantlets are depicted as boxes, output as ellipses.

Our raw data comes in a variety of formats and resolution and sometimes with additional variables not relevant to our analysis. In MOErawdata we take care of this by adjusting the time zones, labels and formatting and by removing all unwanted variables. It quickly became clear that our data contains a number of missing values, which are visualized in MOEplotNAs. To deal with these we interpolate them on the basis of daily and weekly seasonal patters in MOEinterpolation. The interpolated data is aggregated to daily values (which takes care of differing resolutions) and merged in MOEmergedata, which produces a coherent datacube ready for testing and regression purposes. We use the chance to explore our data with timeseries and correlation plots in MOEexploratory. The datacube is then used for testing and regressions in MOEregression which also uses the function `YMDDummies()` from the MOEtimedummies Quantlet for the creation of time dummie variables to account for seasonal patterns in the regression.

3. Data Preparation

In this section we give an account of the data cleaning process. As explained, the order of running the Quantlets is important because each Quantlet produces a new dataset with further processed data, saved as a `.Rdata` file. Before we begin explaining each Quantlet in detail we want to take a look at two basic features of our code that are reproduced in nearly every Quantlet:

Environment Handling Usually the first step when loading a Quantlet is clearing the environment and loading the required packages. If a package is not present in the library, it is installed using `lapply()` and `install.packages()`. At the end the environment is cleared again except for the produced dataframes and variables.

```
# Install and load libraries.
libraries = c("tidyr", "lubridate")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
    install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)
```

Listing 1: This code snippet is used in various other Quantlets on QuantNet.

Working Directory Handling We were unsure how we should go about handling working directories, so we wrote all our Quantlets assuming they would be run from the root of our GitHub repository. If one wishes to run the Quantlets from somewhere else, it is necessary to adjust this manually in our code.

```
# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")
```

Saving Dataframes Generally results of a Quantlet are saved as a `.Rdata` and a `.csv` file to allow for further processing with other Quantlets or (theoretically) with other programs. The `.Rdata` format is especially convenient as it preserves formatting as `POSIXct`, which is lost when saving as `.csv`.

3.1. Cleaning

MOErawdata - Due to the several different data sources, the data needs to be formatted into a more consistent structure. This quantlet reads in the different data sets and formats them using the `lubridate` and `tidyr` packages. After loading in the data, we take the following steps (*although the order of operations may differentiate, depending on the specific variable*):

Remove and Rename Columns Some columns in our raw datasets were not relevant for our analysis such as energy prices in $. The rest of the columns were renamed in a consistent fashion.

Parse Date–Time Data The package `lubridate` is used to coerce the data into a consistent `POSIXct` time format (ISO8601) and to handle the different timezones. The package `tidyr` was used to separate variables that were not directly convertible. For example, each value of the `DEMAND` date–time column comprised a starting ('von') and end ('bis') point of measurements in a 15 minutes interval. Using `tidyr`, the values were separated (see MOErawdata).

```
y        = separate(y, col = TIME, into = c("TIME","bis"), sep = " - ")
y        = subset(y, select = c("TIME","DEM"))
y$TIME = dmy_hm(y$TIME, tz = "UTC")
```

Our data sources used differing time zones, which had to be handled by specifying the right timezone when converting to `POSIXct` and subsequently coercing all date–times to `UTC` using the `with tz()` function from the `lubridate` package (see MOErawdata).

```
df.pun$TIME  = ymd_hm(df.pun$TIME, tz = "UTC")
df.solar$TIME = dmy_hm(df.solar$TIME, tz = "Europe/Brussels")
```

```
df.solar$TIME  = with_tz(df.solar$TIME, tz = "UTC")
```

After those transformations one issue was unresolved. On the day of the switch to daylight saving time (from CET to CEST), duplicate values were being introduced for four quarters of an hour (between 2 and 3 o'clock). The function `fixDlsDups()` was written to take care of that, by searching for duplicate date–time values and shifting them by one hour. This function was applied to variables using CET/CEST format.

```
# Handle a bug that creates duplicate values when daylight saving time changes.
fixDlsDups = function(times, fromLast = TRUE){
    # Searches for duplicate values and subtracts 1 hour. Returns fixed dates.
    dups       = which(duplicated(times, fromLast=fromLast))
    times[dups] = times[dups] - dhours(1)
    return(times)
}
```

Coerce Data into Numeric Values To coerce data into numeric type the functions `as.numeric()` were used or, if the value was of the type factor, the slightly more laborious `as.numeric(levels(x))[x]` was used in order to get the correct value instead of the level of the factor vector. Generally there were many issues, where variables comprised values that made them difficult to coerce. For example the Austrian wind data comprised both, a comma as decimal separator and a point as thousand mark, making coercion difficult. These issues could be resolved by replacing the disruptive elements using the `sub()` function (see MOErawdata).

```
y$‘WIND.MW.AT‘  = as.factor(sub("[:.:]", "", y$‘WIND.MW.AT‘))
y$‘WIND.MW.AT‘  = as.numeric(sub(",", ".",
                             levels(y$‘WIND.MW.AT‘)))[y\$‘WIND.MW.AT‘]
```

Join variables Many datasets were only available for download in set time intervalls. For many variables the raw data therefore consisted in a multitude of data frames, each for a specific time-period. For this reason we wrote functions to execute the above mentioned processes for multiple dataframes (see MOErawdata).

```
select.ATSOLAR = function(x){
  # Selects and cleans the important variables for the ren.AT data
  #
  # Args:
  #   x: Imported raw dataframe
  #
  # Returns:
  #   y: Corrected solar.AT dataframe
  y                 = subset(x, select = names(x)[c(1, 7)])
  names(y)          = c("TIME", "SOLAR.MW.AT")
  y$‘SOLAR.MW.AT‘  = as.numeric(y$‘SOLAR.MW.AT‘)
  y$TIME            = dmy_hms(y$TIME, tz = "Europe/Brussels")
  y$TIME            = fixDlsDups(y$TIME)
  y$TIME            = with_tz(y$TIME, tz = "UTC")
  return(y)
}
```

After this the different data frames of each variable were joined together, to create one coherent variable (see MOErawdata).

```
# Bind dataframes.
df.dm       = rbind(df.dem.2015,df.dem.2016,df.dem.2017,df.dem.2018)
df.solar.AT = rbind(df.solar.AT1, df.solar.AT2,df.solar.AT3,df.solar.AT4)
df.wind.AT = rbind(df.wind.AT1,df.wind.AT2,df.wind.AT3,df.wind.AT4)
```

Save data-frames Save variables as `.csv` and as `.Rdata` file for further processing with other Quantlets.

3.1.1. Plotting Missing Values

MOEplotNAs - After combining our variables into single dataframes, we had the chance to take a first look at our data. We soon realized that there was quite a high number of missing values in our dataset. This was especially problematic as we were going to aggregate hourly and quarter hourly values daily, meaning that a single missing value per day would lead to a whole missing day. To see how big of a problem this actually is, we created a Quantlet that provides a function `DiagMissingValues(df, dlevel=0)` which returns a dataframe of date–times of missing values in `df` using `complete.cases()` (see MOEplotNAs).

```
tp <- subset(df\$TIME, complete.cases(df) == FALSE)
tp <- as.data.frame(tp)
```

It also displays NA–diagnostics when the optional parameter `dlevel` is set to `1` or `2`.

```
> tmp <- DiagMissingValues(df.solar, dlevel=1)
------ NA Diagnostics for 'df.solar' ------
Number of complete cases: 252701 of 254208.
Number of incomplete cases: 1507 (0.593%).
      TIME     FzHertz     Amprion TenneT.TSO Transnet.BW
         0         524         700          9         310
```

Listing 2: The function shows the percentage of missing values and also how the missing values distribute (in this case) over the solar energy producers.

The function was then used to create a histogram of missing values by day for the affected variables in our dataset. Luckily for us the missing values appear most often in bundles, only affecting 2 percent of all days (see MOEplotNAs).

Figure 3: Number of missing values per day in the years 2015–2017 from the raw dataset, colored by variable.

3.2. Interpolation

3.2.1. Purpose

MOEinterpolation - The purpose of this Quantlet is to handle the missing values in our data using interpolation methods. Although we use daily data in our regression, we interpolate the data with the resolution of the raw data (15 Minutes) in order to preserve the most information possible. The only variables that display missing values are solar generation in Germany, demand values for the German-Austrian market, and wind generation in Germany. Thus all other variables do not need to be handled for NA's. As these three variables have some different characteristics, we use different interpolation techniques for each of them.

Interpolation for solar generation in Germany Regarding solar generation, we observed that some missing values happened at night. Therefore, in a first step we built a function which replaces these values by zero. In a second step, we interpolated missing values by their adjacent neighbours, as neighbouring solar generation values should have similar values. Nevertheless, we limited the allowed consecutive NA's to be replaced by this process to 1 Hour intervals (4 quarter-hourly data points), as we think that longer intervals could lead to huge errors (ie. if 12 hours are missing, solar generation during the day might be interpolated with values at night, therefore leading to zero generation). Finally, we interpolated the NA's left, taking into account seasonality. For each missing quarter-hourly value, the solar generation values were averaged at the same time of the day before and the day after. Initially, we operated a regression with seasonal dummies in order to forecast missing values. As the latter yielded some negative forecasted values (because of varying seasonal patterns within the day, due to varying sun exposure lengths), we decided to simplify the interpolation process with the above described averaging process. The described patterns can be seen in Figure 4.

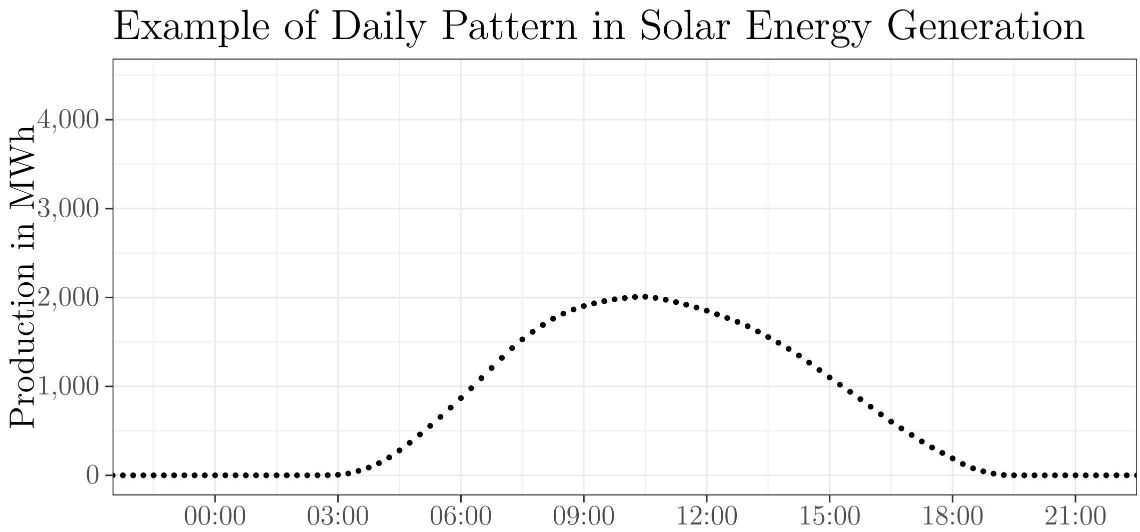


Figure 4: Solar Energy Production on July 12, 2016.

Interpolation for demand on the German-Austrian Market Concerning demand data, we started with a linear interpolation of NA's by their adjacent neighbours, just as for the solar data. The NA's that were left, due to our restriction to one hour of consecutive missing values, were interpolated taking into account seasonality. This process is similar to the one we used for the solar data, except that we needed to take into account the fact that demand also has a seasonality according to the days of the week. Thus we interpolated missing values by averaging the demand values at the same time of the day for the previous and the next non-missing value on the same day of the week. (ie. If demand is missing for the time 12:00 on Monday the 10th, we took the average of demand on Monday the the 3rd and Monday the 17th at 12:00). The described patterns can be seen in Figure 5.

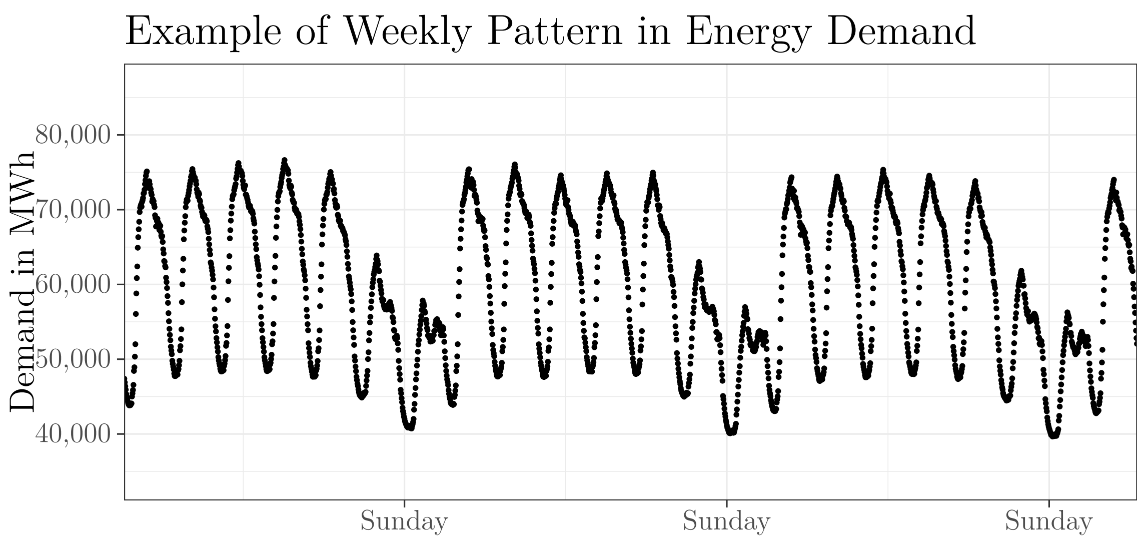


Figure 5: Energy demand during three weeks of July 2016. Note the dips on Saturdays and Sundays. Demand has a strong daily and weekly pattern.

Interpolation for wind generation in Germany We chose to deal with NA's in the Wind data on a daily basis rather than in the small intervals, using a later Quantlet, where the data is already aggregated to daily values. This is due to the fact that NA's always happen for several days in a row, and do not have an identified seasonality. Therefore, it would have been wrong to interpolate the data for multiple days on the basis of the last and the next non-missing quarter hourly value, and we could not use a seasonal pattern to ameliorate our approximation.

3.2.2. Implementation

The code that deals with NA interpolation uses the structured data MOEdata_clean.Rdata, provided by the Quantlet MOErawdata.

Set consistent time frame For the regression we need variables that start and end at the same time. Because the data we have is from different sources, we have many different start- and end-points. Therefore, after loading the data, we defined a function, that can be used on all our variables and sets a consistent time frame for all our data/ variables. (see MOEinterpolation)

Replace NA's occuring at night by zero in solar data for Germany In a first step, we defined two functions "Sunrise.DE" and "Sunset.DE" (see MOEinterpolation) that return the date/time of sunrise and sunset in Berlin for a given date. In the following we will use these functions in order to replace night values by zero for each of the data vectors we have for the four German Transmission System operator (TSO). The data frame that contains our solar generation values for Germany is displayed below:

```
> head(df.solar)
                    TIME FzHertz Amprion TenneT.TSO Transnet.BW
131625 2015-01-01 00:00:00    0       0          0           0
131626 2015-01-01 00:15:00    0       0          0           0
131627 2015-01-01 00:30:00    0       0          0           0
131628 2015-01-01 00:45:00    0       0          0           0
131629 2015-01-01 01:00:00    0       0          0           0
131630 2015-01-01 01:15:00    0       0          0           0
```

Listing 3: dataframe of the German solar data

In the code excerpt below (see MOEinterpolation) we apply a loop that replaces NA's occuring at night for each of the columns of our data frame (below on line 147).
In each iteration, the dates and the missing values for one TSO are stored in the dataframe "Missings" (below on line 155). Subsequently, the sunrise.DE and sunset.DE functions are applied to those dates (line 161 & 176). Therefore, we get two data frames "sunrise.DE.df" and "sunset.DE.df" that contain the times of sunrise and sunset for each missing value (line 167 & 176). Next, we use those two data frames in order to check whether the individual NA's are occuring at night. If yes, their value is replaced by Zero (line 179). Finally, we override the NA's that we have in our xts file "xts.solar" (line 183). (This "xts.solar" had originally been defined by "df.solar"). We use .xts because it facilitates the handling of dates. These files use date/times as index and order the data automatically according to their date/time

```
147 for (TSO in names(df.solar[-1])){
148  # Repeats the following procedure over the 4 colums
149  # FZHertz", "Amprion", "TenneT.TSO", "Transnet.BW"
150
151  if (length(subset(df.solar, is.na(df.solar[,TSO]) == TRUE)[,TSO])>0) {
152    # Tests whether the column has no NA's
153    # as this would lead to an error in the loop
154
155    Missings = subset(df.solar, is.na(df.solar[,TSO]))[,c("TIME", TSO)]
156    # Selects only the rows of df.solar that contain NA's in the column of
157    # the index "TSO"
```

```
    # Was implemented for efficiency, as it would take a significant amount
    # of time to repeat this procedure over all rows

    sunrise.DE.List = lapply(Missings$TIME, Sunrise.DE)
    # Applies the above defined function (Sunrise.DE) on the dates of the
    # missing values for the column "TSO"
    # This will give a list containing two elements per calculated function
    # ( 1 per row of x.missing)

    sunrise.DE.df = do.call(rbind,sunrise.DE.List)
    # Transforms the List into a data.frame

    sunset.DE.List = lapply(Missings$TIME, Sunset.DE)
    # Applies the above defined function (Sunset.DE)on the dates of the
    # missing values for the column "TSO"
    # This will give a list containing two elements per calculated
    # function ( 1 per row of x.missing)

    sunset.DE.df = do.call(rbind,sunset.DE.List)
    # Transforms the List into a data.frame (easier to handle)

    Missings[Missings$TIME < sunrise.DE.df$time | Missings$TIME > sunset.DE.df$time,
        TSO] = as.numeric(0.0)
    # For all values of the df "Nissings", replace NA's by the value 0.0 whenever
    # it is before sunrise or after sunset

    xts.solar[Missings$TIME, TSO]=Missings[,TSO]
    # replaces missing solar values at night by 0 in the xts file containing
    # all solar data

  }
}
```

Listing 4: Replace solar values that happen at night by Zero.

Next Neighbour Interpolation for Solar and Demand in Germany The next neighbour interpolation is a simple linear interpolation of the next and last non-missing value, with a condition on the maximum amount of consecutive missing values to be interpolated. It applies to the NA's that do not have been handled in the previous interpolation step. In this part, we simply applied the function "na.approx" from the zoo.package on both xts files for Demand and Solar in Germany.

Interpolation Taking into Account the Seasonality of the Data for Demand and Solar in Germany This final step handles the NA's left with an averaging process that is based on seasonal patterns. For the Demand values, we implemented a code that follows the following logic: The Demand data is split up according to their day of the week and their time. Therefore, we generate one xts file for each quarter hour - day of the week combination. Then the function "na.approx" of the zoo package is applied to each of those xts files. Therefore, missing values are interpolated linearly between their previous and next non-missing value at the same time of day on the same day of the week. Let's now describe the code more specifically. First, we defined two functions "HM" (line 213-215) and "WeekDay" (line 264-266), that return respectively the time in the format "HH:MM" and the day of week (0 for Sunday, 1 for Monday etc...) for a given date/time. Then we defined the vector "quarter.days" of all quarter hours of the day (line 217-221), and the vector of all the days of the week "day.week" (line 261). Those two vectors are then used in the double loop displayed in the code extract below (see MOEinterpolation). This loop iterates on the days of the week and on the quarter hours of each day. In these iterations xts files for Demand values of each quarter-hour/day of the week combination are stored in the nested list "dm.day.list" (line 282) (One xts for all data for time "00:00" on Monday, one for "00:00" on tuesday etc...). This nested list is used to enable accessing each xts file easily in the defined double-loop, as the list stores for each day of the week, a list of xts files for each quarter hour of the day. Then, on line 287, the function "na.approx" that has been described before, interpolates linearly the NA's for each of the xts files of the nested list. Finally, the interpolated values override the original values in our main xts file for Demand values. (line 296)

```
268 #define list to store split up xts files
269 dm.day.list = list()
270
271 for (Day in day.week){
272 # This double loop splits up the orginal xts into xts files for every quarter
273 # hour and day of the week.(One xts for all data for time "00:00" on Monday,
274 # one for "00:00" on tuesday etc...)This is done so that missing values can be
275 # interpolated on data for the same time and the same day of the week.
276
277   dm.day.list[[Day]]=list()
278   #defining the nested list
279
280   for (quarter in quarters.day ) {
281
282     dm.day.list[[Day]][[quarter]]= xts.dm[HM(index(xts.dm)) == quarter &
283                                         WeekDay(index(xts.dm)) == Day-1]
284      # creates a new xts for each quarter hour for each day of the week
285
286
287     interpolated.values = na.approx(dm.day.list[[Day]][[quarter]],
288                                     na.rm=FALSE,
289                                     maxgap=2)
290     # for every such xts the NA's are interpolated.
291     # ie: a NA on Monday 25th of March at 12:00 is interpolated by the demand
292     # values on Monday the 18th of march at 12:00
293     # and Monday the 1st of April at 12:00
294
```

```
295
296     xts.dm[index(dm.day.list[[Day]][[quarter]])] = interpolated.values
297     #all values of the original xts file get replaced by the interpolated ones
298   }
299
300 }
```

Listing 5: Replace NA's in Demand values considering the seasonal pattern

The interpolation for Solar generation in Germany is very similar to the one for Demand, except that it is slightly simpler. Indeed, as solar generation does not have a seasonal pattern with respect to the day of the week, we apply the same logic as for Demand without needing to use a double loop. In this part we simply generate xts files for every quarter hour of the day through a simple loop. The NA's get interpolated by the previous and next non-missing Solar data at the same time of the day (a missing value at 12:00 on the 13th of May gets interpolated by the values at 12:00 on the 12th and on the 14th). As this code is simpler but similar to the one described above, we will not get into its detailed implementation (for further detail see line 209-254).

Finally, once this interpolation in multiple steps has been performed for Solar and Demand, we have two xts files, "xts.solar" and "xts.demand" without missing values. These two files get converted back to the data frame format, so that the merging can be performed in the Quantlet MOEmergedata

3.3. Aggregation and Merging

MOEmergedata - This Quantlet further processes the data from MOErawdata and MOEinterpolation utilizing the `lubridate` package and merges them into one dataframe that can then be used for the regression. After loading the data we take the following steps (the order of operations may vary, depending on the specific dataset/variable):

Aggregating to Daily Values Due to variation in data sources, the data was not only formatted differently but also measured in different intervals. Some variables were measured in 15 minutes intervals, others hourly. For the analysis we needed daily values. Using the aggregate function we calculated the required values (see MOEmergedata). The variables in 15 minute intervals had to be computed differently because the data was required in MW/h. One could not simply add up the values for a day but has to divide the summed values by four, in order to get the correct MW/h value (see MOEmergedata).

Combine Austrian and German Renewable Production The data for the German renewable production was split up by producers. To combine the German and Austrian renewable production we sum up the values for the German producers using the `rowSums()` function and add the Austrian renewable production (see MOEmergedata).

Rename and Join Variables In the last step of the data processing we combined all the variables into one dataframe, which can then be used from all further Quantlets (see MOEmergedata).

3.4. Exploring the Data

MOEexploratory - After the cleaning and processing of the data, we explored our data with time series and correlation plots, in order to better understand the structure and characteristics of the data we use. We load the data set from the Quantlet MOEmergedata and create plots using the `ggplot2` package. We also use the packages `tidyr` to turn our data into `tidy`–data, as this allows for more natural plotting with `ggplot2`. We created a faceted timeseries plot (8) and a correlation plot (9). As they are quite large, they can be found in the appendix A.1. We also create the trend plots (4 and 5) we have seen earlier.

4. Regression and Tests

4.1. Creating Time Dummy Variables

4.1.1. Purpose

MOEtimedummies As we have seen in the theoretical part, we specify a regression in which we control for the seasonality of the price data. This control is performed by specifying dummy variables for the Years, the Months and the Days of the week. As we did not find an appropriate package to deal with this, we decided to code a function of our own that extends our data by a dummy variable matrix. This function is specified in the MOEtimedummies.R script. This function "YMDDummy" takes an xts file as an input and adds dummy columns for each year, each month and each day of the week on its right side (except for one of each category in order to avoid the "dummy trap") As before, we use the xts format in order to use its capabilities for time series.

4.1.2. Implementation

In a first step, the minimum year, the maximum year and a vector of all different years we have in our data are defined (in this case 2015, 2016 and 2017). These correspond to the variables "Year.min", "Year.max", "Year.Vector". The variables "Year.min.number" and "Year.max.number" are the variables "Year.min" and "Year.max" in numeric format. (line 36-40)

Then, as we can see in the code excerpt below (see MOEtimedummies), we define an empty matrix with the number of rows of the input xts file, and with as many columns as there are different years in the xts input file (line 80). After that, we use a loop that iterates as many times, as there are different years(ie. 3 times if we have data going from 2015 to 2017). In each iteration, the vector that contains the year for each data point of the input xts file ("ReturnYear(FullDat.xts)") is compared logically to the value of the year of the dummy column (line 84-86). For example, in the iteration corresponding to the year 2015, values of the column "2015" are set to "TRUE" if the corresponding dates in the input xts file are in the year 2015, they are set to "FALSE" if not. These values are stored in the matrix "Year.Dummy.matrix".

```
80  NumYears                   = Year.max.number-Year.min.number+1
81  Year.Dummy.matrix          = matrix(,nrow = Length , ncol=NumYears)
82  colnames(Year.Dummy.matrix) = Year.Vector
83
84  for (i in 1:length(Year.Vector)) {
85      Year.Dummy.matrix[,i] = ReturnYear(FullDat.xts) == Year.Vector[i]
86  }
```

Listing 6: Generate the dummy variable matrix for Years

In a very similar fashion as for years, we build up a dummy matrix for months and one for days of the week (respectively in lines 89-99 and lines 102-122). Thus we obtain three matrices "Year.Dummy.matrix", "Month.dummy.matrix" and "Day.Dummy.matrix" that all have as many rows as the input xts file, and that display the value "TRUE" if the corresponding date of the input xts file matches the corresponding year, month or day of the week of the corresponding dummy column.

Then, we bind the three matrices per column line 128. In this very same step, one of each dummy column type is deleted in order to avoid the "dummy trap". The user can specify which one should be deleted using the function's options. Then we convert the resulting matrix to xts format line 131 and convert the boolean values into numerical 1's and 0's (respectively for "TRUE" and "FALSE") line 132. . Finally, we bind the dummy matrix on the right side of the input xts file line 137

4.2. Implementing the Regression and Tests

MOEregression - This final part of our code, is the one, where we use our final aggregated, cleaned and merged data (MOEdata_merge.Rdata) in order to replicate the Italian paper on the German-Austrian electricity Market, as described in section Introduction and Theory. This part deals with the code implementation. Results will be displayed and discussed in the section Empirical Study Results.

In a first step, our final data frame is loaded and transformed into an xts file, so that our dummy matrix function "YMDDummy" can be applied on it. As mentionned before, wind generation data is interpolated linearly here on a daily basis line 74. Then, the xts table is converted into the ts format so that the usual functions for time series can be applied on it. By the way, the final demand, solar and wind generation data are transformed from daily total demand/generation in [MWh] into an hourly average for each day in [GW] line 93. (This part helps ameliorating the interpretation of the regression results, by yielding coefficients without exponents). In a first step, the Dickey-Fuller test and the Philipps-Perron test for trend-stationarity are performed on the individual variables in our data (Price, Demand, Solar generation, and Wind generation) line 107 - 130. The corresponding functions are the function "adf.test" and "pp.test" from the package "tseries". The p-values of those tests with H_0:" The time-serie has a unit root" are then stored in the Latex table ADFandPPTEST.tex, defined on lines 133 - 142.

In the next step, a simple OLS with the Model specification is run with the function "tslm", of the package "forecast" line 148 - 154. Its output is then used to perform the Durbin-Watson test for autocorrelation of the error terms and the Breusch-Pagan test for heteroscedasticity. (functions "dwtest" and "bptest" of the package "lmtest") line 154 - 166. P-values are finally stored automatically in the Latex table OLS_DWandBPTEST.tex defined on line 176 - 184. Moreover, the function "acf2" of the package "astsa" is used to produce the plot of the auto-correlation (ACF) and partial auto-correlation (PACF) functions of the error terms PACF_OLS.jpg. Finally, the Prais-Winsten regression is run with the command "prais.winsten" of the package "prais", the Durbin-Watson test is run again on the new error terms and the new ACF and PACF functions are plotted lines 191

- 254. Coefficient results are stored automatically in the Latex table PraisWinstenReg-Coefficients.tex, the coefficients of determination and the results of the Durbin-Watson test are stored in the separate table PraisWinstenGoFDW.tex. The new ACF and PACF are stored automatically in the .jpeg file. PACF_PraisWinsten.jpg

5. Empirical Study Results

Tests for unit roots in the regression variables As can bee seen in the table below, our results regarding stationarity are pretty similar to the ones of Clò, Cataldi, and Zoppoli. When we control for a constant and a deterministic time trend, the hypothesis that Solar Gen. has a unit root cannot be rejected, while it can for El. price, El. Demand and Wind Gen (at a 5% significance level). Nevertheless, when we take into account possible autocorrelations and heteroscedasticity when we use the Philipps-Perron test, the hypothesis that the variables have a unit root is rejected for all variables (at a 5% significance level). Therefore, as Clò, Cataldi, and Zoppoli did, we shall keep our variables as they are.

	A. Dickey Fuller	Philipps-perron
El. Price	0.01	0.01
El. Demand	0.01	0.01
Solar Gen.	0.32	0.01
Wind Gen.	0.01	0.01

Table 1: p-values of unit root tests (with time trend and constant)
H_0: The series has a unit root

OLS and auto-correlation of the error terms As can be seen in the table below, the Durbin-Watson test applied on the OLS shows that the H_0: " The error terms are not auto-correlated" is rejected at a 1% significance level. Thus, this autocorrelation shall be modelled. The Breusch-Pagan test rejects the Null-Hypothesis that the error-terms are homoscedastic, thus heteroscedasticity must be taken into account in further steps.

As can be seen below, the PACF of the error terms in the OLS displays a sharp cut-off after the first lag. This could be an indication for an AR(1) process. Nevertheless, when we look more closely, there are later lags that display a PACF that is significantly

	Durbin-Watson	Breusch-Pagan
p-value	0.00	0.00

Table 2: Tests for auto-correlation and heteroscedasticity
H_0^{DW}: The error terms are not auto correlated
H_0^{BP}: The error terms are homoscedastic

different from 0 at later lags (at a 5% significance level). Furthermore, the ACF does not display the geometrical decay that is typical for an AR(1) process. Although, an AR(1) modelisation of the error terms might not be the best possible modellisation of the error terms here, we shall proceed with the same methodology as in the original paper.

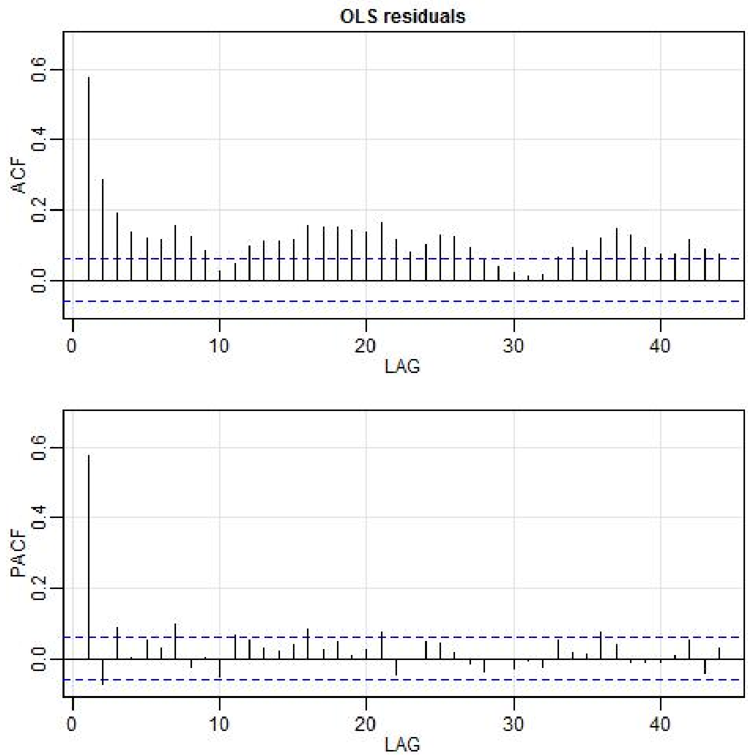


Figure 6: ACF and PACF of the OLS

Prais-Winsten regression The estimation of the Prais-Winsten regression that models the error terms by an AR(1) process can be found below. It can be seen that the coefficients for the variables Demand, Solar Gen., and Wind Gen. have a coherent sign and magnitude, and they are all significantly different from Zero at a 5% significance level. A one GWh increase of the hourly average of daily demand increases the electrical price by 0,2 EUR/MWh in the German-Austrian market (Compared with 2,26 EUR/MWh in the italian paper). A one GWh increase of the hourly average of the daily Solar Gen.

decreases the electrical price by 0,49 EUR/MWh. The same increase in the daily Wind Gen. decreases the electrical price by 1,20 EUR/MWh (Compared with decreases of 2,58 EUR/MWh and 4,19 EUR/MWh in the italian market). There are probably numerous reasons for these magnitude differences in the coefficients in the German-Austrian and Italian market. One of them is probably that the German-Austrian market is much bigger than the Italian one. Thus a similar increase in Demand, Solar Gen. or Wind Gen. is relatively smaller in the German-Austrian market and might have a smaller impact on the electricity price. For instance, the Demand in the German-Austrian market is roughly twice the one in Italy (see: electricity consumption statistics IEA)

	Estimate	Std. Error	t value	Pr(>|t|)
Demand	0.20	0.02	9.41	0.00
Solar Gen.	-0.49	0.21	-2.31	0.02
Wind Gen.	-1.20	0.12	-10.13	0.00
Year 2016	-2.68	1.24	-2.15	0.03
Year 2017	1.41	1.26	1.12	0.26
February	-4.77	2.17	-2.20	0.03
March	-5.71	2.30	-2.49	0.01
April	-1.65	2.47	-0.67	0.50
May	-3.37	2.53	-1.33	0.18
June	-0.84	2.56	-0.33	0.74
July	0.24	2.52	0.10	0.92
August	-0.56	2.51	-0.22	0.83
September	0.37	2.39	0.16	0.88
October	2.15	2.29	0.94	0.35
November	-0.21	2.26	-0.09	0.93
December	-3.05	2.17	-1.41	0.16
Monday	2.55	1.15	2.22	0.03
Tuesday	2.85	1.36	2.10	0.04
Wednesday	2.28	1.40	1.63	0.10
Thursday	1.56	1.39	1.12	0.26
Friday	1.76	1.30	1.36	0.17
Saturday	1.97	0.67	2.94	0.00
Intercept	-13.37	5.07	-2.63	0.01
rho (AR1)	0.62		26.17	

Table 3: Prais-Winsten regression results

R^2	$Adj.R^2$	Durbin-Watson (p value)
0.81	0.81	0.04

Table 4: Prais-Winsten regression results

To finish with, we shall assess whether the specified Prais-Winsten regression is suitable for our data. Unfortunately, as we can see in the table above, the Durbin-Watson test shows that the Null-Hypothesis: " There is no auto-correlation of the error terms" is rejected at a 5% significance level. This result can be visualized in the ACF and PACF function plots displayed below, where it can be observed that both functions cross the 5% significance level line at numerous lags. Therefore, the AR(1) modelling of the error terms is not appropriate for our case. If we want to improve our results, we should investigate a way to deal with the autocorrelation differently. Maybe by modeling the error terms with an ARMA process.

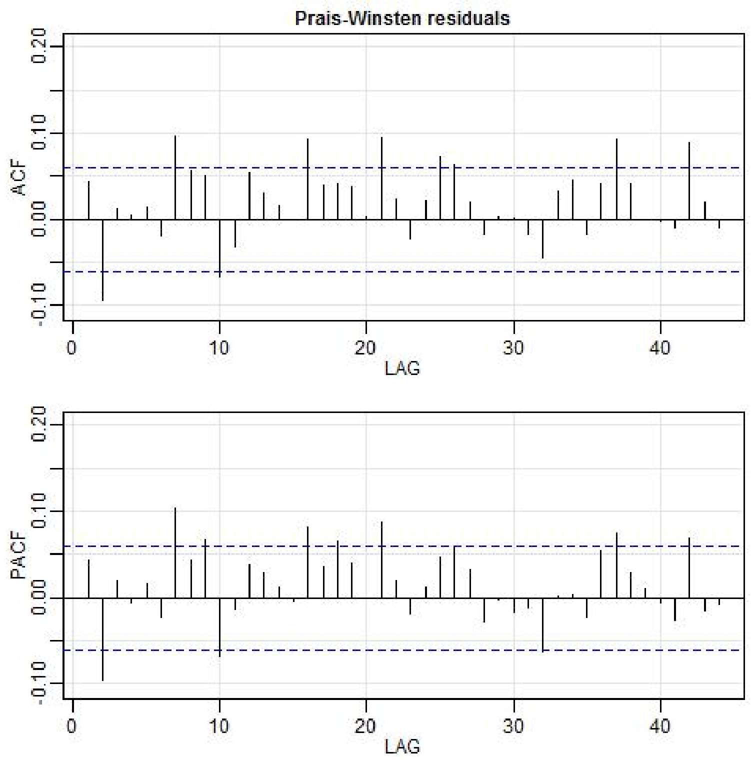


Figure 7: ACF and PACF of the Prais-Winsten regression

6. Conclusion

In this seminar paper, we have gone through all the steps of gathering, formatting, and cleaning data in order to replicate Clò, Cataldi, and Zoppoli 2015 for the German-Austrian electricity market. We find that renewable energy production had a significant negative effect on the energy price in the years 2015–2017. This effect has plausible magnitude and is in accordance with the theory. However, applying Clò, Cataldi, and Zoppoli 2015's methodology to our data was not strictly adequate as auto-correlations remain in the final regression. In order to improve our results, the auto correlation would have to be modelled more precisely and gas price data should be used in the regression. Also, our dataset was quite small, only encompassing three years.

A. Appendix

A.1. Exploratory Plots

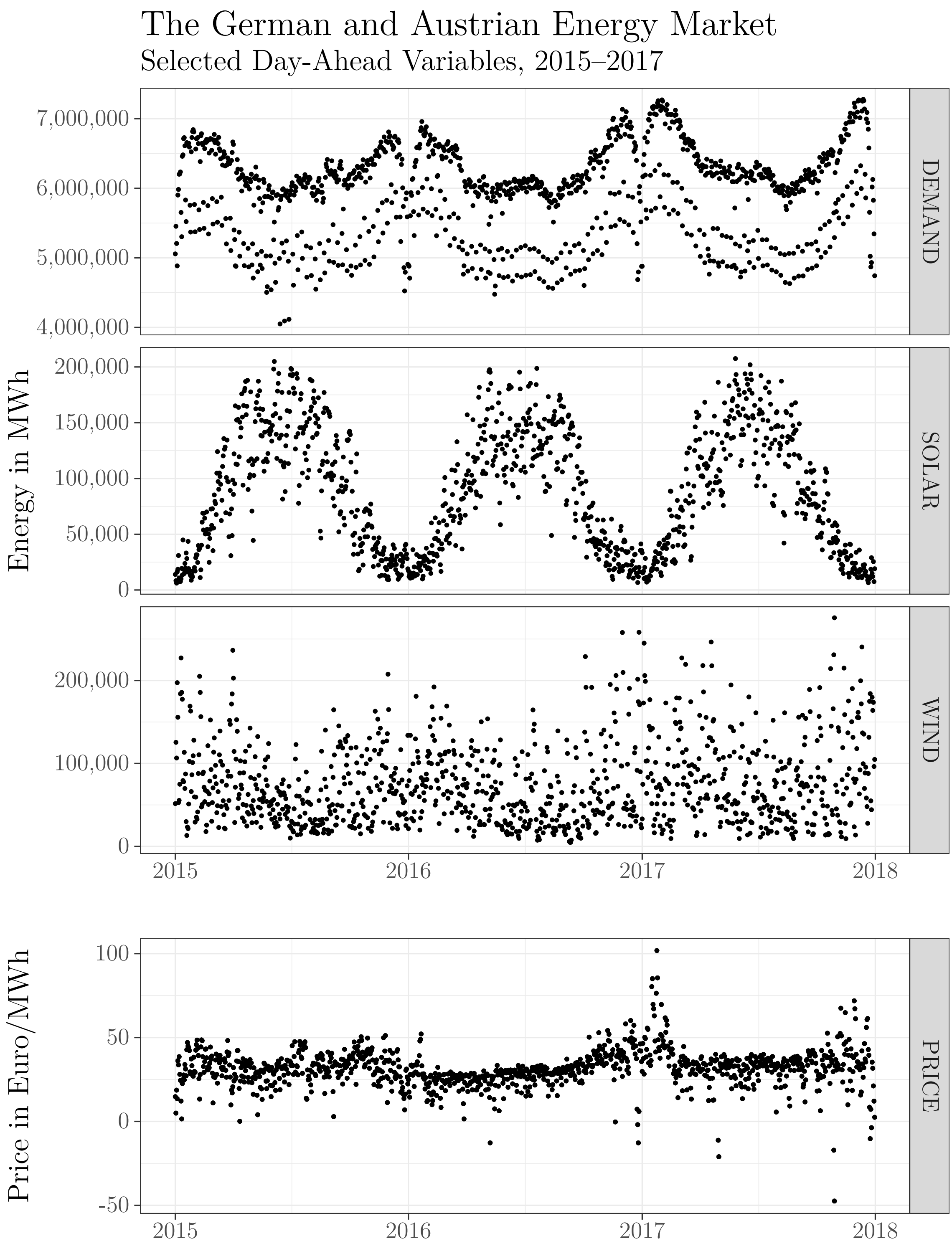


Figure 8: Time series plot of selected energy market variables during the years 2015–2017.

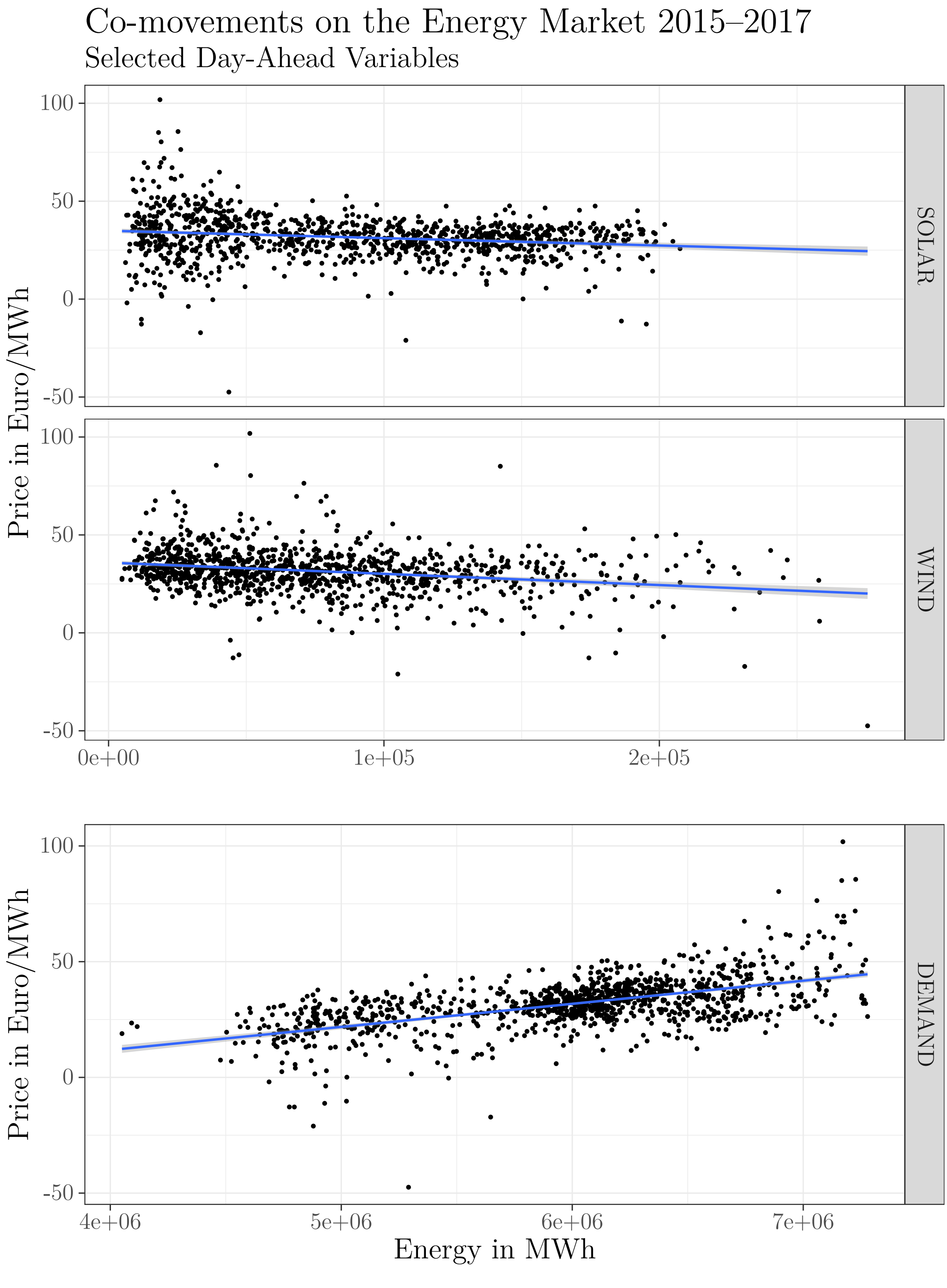


Figure 9: Correlations between energy price and renewable energy production as well as energy demand during the years 2015–2017.

A.2. Full Source Code Listing

A.2.1. MOEqletflow.R

```r

###############################################################################
####   MOEexploratory.R  ###################################################
###############################################################################
#
# Creates flowchart of MOE quantlet structure.
#
# Output: MOEchart_qlet.tex     - flowchart in .tex format
#         MOEchart_qlet.pdf     - flowchart in .pdf format
#
###############################################################################

# Clear all variables.
rm(list = ls(all = TRUE))
graphics.off()

# Install and load libraries.
libraries = c("DiagrammeR")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
  install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)



###############################################################################
####   0.  SET WORKING DIRECTORY  ##########################################
###############################################################################

####   ATTENTION: Working directory is assumed to be the root of the MOE
####   repository, not the MOEmergedata Quantlet subdirectory!!!


# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")



###############################################################################
####   1.  CREATE FLOWCHART   ###############################################
###############################################################################


chart_qlet = "

digraph qlets {

    graph [layout = neato,
          overlap = true,
          outputorder = edgesfirst]

    # add note statements
    node [shape = box]
```

```
    A [pos = '-3,0!', label = ' MOErawdata ']
    B [pos = '-1,0!', label = ' MOEinterpolation ']
    C [pos = '1,0!', label = ' MOEmergedata ']
    D [pos = '3,0!', label = ' MOEregression ']
    F [pos = '3,1!', label = ' MOEtimedummies ']
    P1 [pos = '-3,1!', label = ' MOEplotNAs ']
    P2 [pos = '1,1!', label = ' MOEexploratory ']

    node [shape = ellipse]
    O [pos = '5,0!', label = 'Regression\nOutput']
    O1 [pos = '-3,2!', label = ' MOEplot_na ']
    O2 [pos = '0,2!', label = ' MOEplot_expl ']
    O3 [pos = '2,2!', label = ' MOEplot_corr ']

    # add edge statements
    edge [arrowhead = diamond, headport = 'w', tailport = 'e']
    A -> B;
    B -> C;
    C -> D;

    edge [arrowhead = diamond, headport = 's', tailport = 'n']
    F -> D [headport = 'n', tailport = 's'];
    A -> P1;
    C -> P2;

    edge [arrowhead = arrow, headport = 'w', tailport = 'e']
    D -> O;

    edge [arrowhead = arrow, headport = 's', tailport = 'n']
    P1 -> O1;
    P2 -> O2;
    P2 -> O3;

}
"

grViz(chart_qlet)
```

A.2.2. MOErawdata.R

```

################################################################################
####   MOErawdata.R   ######################################################
################################################################################
#
# Energy market data is read in using 'readr'and formatted using the
# 'lubridate' and 'tidyr' packages. Cleaned <dataframes> are saved as .csv and
# .Rdata files. Working directory may need to be set corectly (see section 0).
#
# Input: Data '.csv' files from the 'input' subdirectory.
#
# Output: MOEdata_clean_csv/<variable>.csv  - data in table form
#         MOEdata_clean.Rdata               - data in Rdata form
#
################################################################################

# Clear all variables.
rm(list = ls(all = TRUE))
```

```
graphics.off()

# Install and load libraries.
libraries = c("tidyr", "lubridate")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
    install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)



################################################################################
####   0.  SET WORKING DIRECTORY  ##########################################
################################################################################
####   ATTENTION: Working directory is assumed to be the root of the MOE
####   repository, not the MOErawdata Quantlet subdirectory!!!


# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")



################################################################################
####   1.  READ ENERGY MARKET DATA  #########################################
################################################################################


# READ PRICE DATA:
#   Units:     UTC, EUR
#   Delim:     ,
#   Decimal:   .
#   Enclosed:  none
#   Header:    TRUE
df.pun.0       =  read.csv("MOErawdata/inputs/price_elspot.csv")

# READ DEMAND DATA:
#   Format:    UTC, MW
#   Delim:     ,
#   Decimal:   none
#   Enclosed:  ""
#   Header:    TRUE
df.dem.2015.0  = read.csv("MOErawdata/inputs/demand_2015_entsoe.csv")
df.dem.2016.0  = read.csv("MOErawdata/inputs/demand_2016_entsoe.csv")
df.dem.2017.0  = read.csv("MOErawdata/inputs/demand_2017_entsoe.csv")
df.dem.2018.0  = read.csv("MOErawdata/inputs/demand_2018_entsoe.csv")

# READ SOLAR.DE, WIND.DE DATA:
#   Units:     CET/CEST, MW
#   Delim:     ;
#   Decimal:   ,
#   Enclosed:  ""
#   Header:    TRUE
df.solar.D     =  read.csv2("MOErawdata/inputs/solar_DE_netztransp.csv")
df.wind.D      =  read.csv2("MOErawdata/inputs/wind_DE_netztransp.csv")

# READ RENEWABLES.AT (SOLAR + WIND) DATA:
#   Units:     CET/CEST, MW
```

```
#   Delim:     ;
#   Decimal:   ,
#   Enclosed:  none
#   Header:    FALSE
df.ren.AT.2015 = read.csv2("MOErawdata/inputs/renew_AT_2015.csv", header = F)
df.ren.AT.2016 = read.csv2("MOErawdata/inputs/renew_AT_2016.csv", header = F)
df.ren.AT.2017 = read.csv2("MOErawdata/inputs/renew_AT_2017.csv", header = F)
df.ren.AT.2018 = read.csv2("MOErawdata/inputs/renew_AT_2018.csv", header = F)



################################################################################
####   2.  CLEAN AND FORMAT  ###############################################
################################################################################


################################################################################
####   2a. SINGLE DATAFRAMES (df.pun, df.solar, df.wind) ##################
################################################################################

# Remove unwanted columns. Merge date and hour columns. Columns are selected by
# their position and not by their name to prevent parsing errors. (mpff)
# Example: Use 'select = names(df)[c(1)]' instead of 'select = "Date"'.
df.pun   = subset(df.pun.0, select = names(df.pun.0)[c(1,5)])

df.solar = subset(df.solar.D, select = names(df.solar.D)[-3])
df.solar = unite(df.solar, TIME, names(df.solar)[c(1,2)], sep = " ")

df.wind  = subset(df.wind.D, select = names(df.wind.D)[-3])
df.wind  = unite(df.wind, TIME, names(df.wind)[c(1,2)], sep = " ")

# Change column names.
names(df.pun)  = c("TIME", "PUN")
names(df.solar) = c("TIME", "FzHertz", "Amprion", "TenneT.TSO", "Transnet.BW")
names(df.wind) = c("TIME", "FzHertz", "Amprion", "TenneT.TSO", "Transnet.BW")

# Format as POSIXct. (Beware of differenct timezones in raw data!)
df.pun$TIME    = ymd_hm(df.pun$TIME, tz = "UTC")
df.solar$TIME  = dmy_hm(df.solar$TIME, tz = "Europe/Brussels")
df.wind$TIME   = dmy_hm(df.wind$TIME, tz = "Europe/Brussels")

# Handle a bug that creates duplicate values when daylight saving time changes.
fixDlsDups = function(times, fromLast = TRUE){
    # Searches for duplicate values and subtracts 1 hour. Returns fixed dates.
    dups       = which(duplicated(times, fromLast=fromLast))
    times[dups] = times[dups] - dhours(1)
    return(times)
}

# Apply subroutine.
df.solar$TIME  = fixDlsDups(df.solar$TIME)
df.wind$TIME   = fixDlsDups(df.wind$TIME)

# Convert to UTC.
df.solar$TIME  = with_tz(df.solar$TIME, tz = "UTC")
df.wind$TIME   = with_tz(df.wind$TIME, tz = "UTC")


```

```
################################################################################
####   2b. MULTIPLE DATAFRAMES (df.dm, df.solar.AT, df.wind.AT) ###########
################################################################################

################################################################################
####   DEFINE SUBROUTINES  #################################################

select.ATSOLAR = function(x){
  # Selects and cleans the important variables for the ren.AT data
  #
  # Args:
  #   x: Imported raw dataframe
  #
  # Returns:
  #   y: Corrected solar.AT dataframe
  y                 = subset(x, select = names(x)[c(1, 7)])
  names(y)          = c("TIME", "SOLAR.MW.AT")
  y$`SOLAR.MW.AT` = as.numeric(y$`SOLAR.MW.AT`)
  y$TIME            = dmy_hms(y$TIME, tz = "Europe/Brussels")
  y$TIME            = fixDlsDups(y$TIME)
  y$TIME            = with_tz(y$TIME, tz = "UTC")
  return(y)
}

select.ATWIND = function(x){
  # Selects and cleans the important variables for the ren.AT data
  #
  # Args:
  #   x: Imported raw dataframe
  #
  # Returns:
  #   y: Corrected wind.AT dataframe
  y                 = subset(x, select = names(x)[c(1,5)])
  names(y)          = c("TIME", "WIND.MW.AT")
  y$`WIND.MW.AT`  = as.factor(sub("[:.:]", "", y$`WIND.MW.AT`))
  y$`WIND.MW.AT`  = as.numeric(sub(",", ".", levels
                                (y$`WIND.MW.AT`)))[y$`WIND.MW.AT`]
  y$TIME            = dmy_hms(y$TIME, tz = "Europe/Brussels")
  y$TIME            = fixDlsDups(y$TIME)
  y$TIME            = with_tz(y$TIME, tz = "UTC")
  return(y)
}


select.DEM = function(x) {
  # Selects the important variables for the demand data
  # Also checks if variable is factor or not
  # Args:
  #   x: Imported raw dataframe
  #
  # Returns:
  #   y: Selection of demand dataframes
  y        = subset(x, select = names(x)[c(1,2)])
  names(y) = c("TIME", "DEM")
  y        = separate(y, col = TIME, into = c("TIME","bis"), sep = " - ")
  y        = subset(y, select = c("TIME","DEM"))
  y$TIME   = dmy_hm(y$TIME, tz = "UTC")
  if (class(y$DEM) == "factor") {
```

```
    y$DEM  = suppressWarnings(as.numeric(levels(y$DEM)))[y$DEM] # suppress NA
    return(y)
  } else {
    y$DEM  = as.numeric(y$DEM)
    return(y)
  }
}

################################################################################
####   APPLY SUBROUTINES  ###################################################

df.solar.AT1   = select.ATSOLAR(df.ren.AT.2015)
df.solar.AT2   = select.ATSOLAR(df.ren.AT.2016)
df.solar.AT3   = select.ATSOLAR(df.ren.AT.2017)
df.solar.AT4   = select.ATSOLAR(df.ren.AT.2018)

df.wind.AT1    = select.ATWIND(df.ren.AT.2015)
df.wind.AT2    = select.ATWIND(df.ren.AT.2016)
df.wind.AT3    = select.ATWIND(df.ren.AT.2017)
df.wind.AT4    = select.ATWIND(df.ren.AT.2018)

df.dem.2015    = select.DEM(df.dem.2015.0)
df.dem.2016    = select.DEM(df.dem.2016.0)
df.dem.2017    = select.DEM(df.dem.2017.0)
df.dem.2018    = select.DEM(df.dem.2018.0)



################################################################################
####   3. BIND AND SAVE DATAFRAMES ########################################
################################################################################


# Bind dataframes.
df.dm       = rbind(df.dem.2015,df.dem.2016,df.dem.2017,df.dem.2018)
df.solar.AT = rbind(df.solar.AT1, df.solar.AT2,df.solar.AT3,df.solar.AT4)
df.wind.AT = rbind(df.wind.AT1,df.wind.AT2,df.wind.AT3,df.wind.AT4)

# Save dataframes as '.Rdata' file for easy read-in in R.
save(df.pun, df.solar, df.solar.AT, df.wind, df.wind.AT, df.dm,
     file="MOErawdata/MOEdata_clean.Rdata"
     )

# Save dataframes as '.csv' files for use with other software.
df.list     = list(df.pun, df.solar, df.solar.AT,
              df.wind, df.wind.AT, df.dm)

df.names  = c("df_pun.csv", "df_solar.csv", "df_solar_AT.csv",
                "df_wind.csv", "df_wind_AT.csv", "df_dm.csv")

lapply(1:length(df.list),
      function(i) write.csv((df.list[i]),
                          file = paste0("MOErawdata/MOEdata_clean_csv/",
                                        df.names[i])))


################################################################################
####   4. CLEAN UP ENVIRONMENT ############################################
```

```
################################################################################


rm(list=ls()[! ls() %in% c("df.pun",
                           "df.solar",
                           "df.solar.AT",
                           "df.wind",
                           "df.wind.AT",
                           "df.dm"
                           )])
```

A.2.3. MOEplotNAs.R

```

################################################################################
####   MOEplotNAs.R   ########################################################
################################################################################
#
# Creates exploratory plots of the NA-value structure in the energy
# market variables.
#
# Input: '.Rdata' file from the 'MOErawdata' Quantlet.
#
# Output: MOEplot_na.tex     - plot in .tex format
#         MOEplot_na.pdf     - plot in .pdf format
#
################################################################################

# Clear all variables.
rm(list = ls(all = TRUE))
graphics.off()

# Install and load libraries.
libraries = c("ggplot2", "tikzDevice", "scales", "lubridate")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
    install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)



################################################################################
####   0.  SET WORKING DIRECTORY ###########################################
################################################################################
####   ATTENTION: Working directory is assumed to be the root of the MOE
####   repository, not the MOErawdata Quantlet subdirectory!!!


# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")



################################################################################
####   1.  LOAD ENERGY MARKET DATA ########################################
################################################################################


```

```
load("MOErawdata/MOEdata_clean.Rdata")



################################################################################
####   2. DEFINE FUNCTIONS ###################################################
################################################################################


DiagMissingValues <- function(df, dlevel = 0) {
  # Checks for NA values in dataframe and prints information.
  #
  # Args:
  #   df: The dataframe that will be checked for missing values.
  #   dlevel: Print more detailed information. 0: No information.
  #           1: General information. 2: Information on columns.
  #
  # Returns:
  #   A <dataframe> of timepoints of incomplete cases in df.

  name <- deparse(substitute(df)) # get name of dataframe

  r <- is.na(df)
  cc <- complete.cases(df)

  if (dlevel >= 1) {
    # dlevel 1 diagnostics: general information.
    cat(sprintf("------ NA Diagnostics for '%s' ------\n", name))
    cat(sprintf("Number of complete cases: %i of %i.\n",
                sum(cc), nrow(df) ))
    cat(sprintf("Number of incomplete cases: %i (%.3f%%).\n",
                (nrow(df)-sum(cc)), (1-sum(cc)/nrow(df))*100) )
    print(apply(r, 2, sum))
  }

  if (dlevel >= 2) {
    # dlevel 2 diagnostics: information per column.
    i <- 1
    for (col in names(df)){
      cat(sprintf("NA's in at least %i columns: %i\n",
                  i, sum(apply(r, 1, sum) >= i) ))
      i <- i+1
    }

  }

  # Get timepoints of incomplete cases and output dataframe
  tp <- subset(df$TIME, complete.cases(df) == FALSE)
  tp <- as.data.frame(tp)
  names(tp) <- "TIME"

  return(tp)
}



################################################################################
####   3. CREATE NA PLOT ###################################################
```

```
###############################################################################


###############################################################################
####   CREATE DATASET

df.na.dm          = data.frame(DiagMissingValues(df.dm), "DEMAND")
names(df.na.dm)   = c("TIME", "SOURCE")

df.na.solar       = data.frame(DiagMissingValues(df.solar), "SOLAR")
names(df.na.solar) = c("TIME", "SOURCE")

df.na.wind        = data.frame(DiagMissingValues(df.wind), "WIND")
names(df.na.wind) = c("TIME", "SOURCE")

df.na.raw         = rbind(df.na.dm,df.na.solar,df.na.wind)

# Create new column for faceting by YEAR
df.na.raw$YEAR = format(df.na.raw$TIME, "%Y")

# Define year vector for relevant years
years = c("2015", "2016", "2017")

# Remove irrelevant years
df.na.raw = df.na.raw[df.na.raw$YEAR %in% years,]

# Change year to dummy year
year(df.na.raw$TIME)  = 2015


###############################################################################
####   CREATE NA PLOT

plot_na = ggplot(df.na.raw, aes(x = TIME)) +
    geom_histogram(aes(fill = SOURCE), alpha = 0.8, binwidth = 24*3600,
                 position = "stack") +
    labs(x = "Date", y = "NAs per day") +
    ggtitle(label = "Missing Values in the Dataset") +
    scale_x_datetime(limits = c(as.POSIXct('2015-01-01 00:00:00'),
                             as.POSIXct('2015-12-31 23:45:00')),
                   labels = date_format("%b"),
                   expand = c(0,0)) +
    facet_grid(YEAR ~ .)


###############################################################################
####   4.  SAVE PLOTS AS TEX FILE   #######################################
###############################################################################


# Save explorative plot as .tex file
tikz(file = "MOEplotNAs/MOEplot_na.tex", width = 8, height = 4)
plot(plot_na)
dev.off()

# Save explorative plot as .pdf file
pdf("MOEplotNAs/MOEplot_na.pdf", width = 8, height = 4)
plot(plot_na)
```

```
dev.off()



################################################################################
####   5. CLEAN UP ENVIRONMENT  ############################################
################################################################################


rm(list=ls()[! ls() %in% c("df.pun",
                           "df.solar",
                           "df.solar.AT",
                           "df.wind",
                           "df.wind.AT",
                           "df.dm",
                           "DiagMissingValues",
                           "plot_na"
                           )])
```

A.2.4. MOEinterpolation.R

```

################################################################################
####   MOEinterpolation.R  #################################################
################################################################################
#
# This code deals with NA's for Solar electricity production
# and electrical demand.
# The missing values in the wind generation data are not handled here, as they
# they will be handled on a daily basis in a later step.
#
#
# Input: 'MOEdata_clean.Rdata' from the 'MOErawdata' Quantlet.
#
# Ouput: MOEdata_interp_csv/<variable>.csv - data in table form
#        MOEdata_interp.Rdata               - data in Rdata form
#
################################################################################


# Clear all variables.
rm(list = ls(all = TRUE))
graphics.off()

# Install and load libraries.
libraries = c("xts", "StreamMetabolism", "lubridate")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
  install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)

# Set default time to "UTC"
Sys.setenv(TZ = "UTC")



################################################################################
####   0.  SET WORKING DIRECTORY  ##########################################
```

```
###############################################################################

####   ATTENTION: Working directory is assumed to be the root of the MOE
####   repository, not the MOEmergedata Quantlet subdirectory!!!


# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")



###############################################################################
####   1.  LOAD CLEAN DATA  ################################################
###############################################################################

load("MOErawdata/MOEdata_clean.Rdata")



###############################################################################
####   2.  MATCH TIMEFRAMES  ###############################################
###############################################################################


###############################################################################
####   DEFINE SUBROUTINES  #################################################

time.FRAME = function(x) {
    # Chooses Time frame for all variables
    #
    # Args:
    #   x: Imported dataframe
    #
    # Returns:
    #   y: Dataframe with right time frame
    #
    # TODO: Adjust timeframe dynamically.
    #
    start.d    = ymd_hm("2015-01-01 00:00")
    stop.d     = ymd_hm("2017-12-31 23:00")
    ind.start  = which(x$TIME == start.d)
    ind.stop   = which(x$TIME == stop.d)
    ind        = (ind.start: ind.stop)
    y          = x[ind, ]
    return(y)
}


###############################################################################
####   APPLY SUBROUTINES  ##################################################

# Match timeframes.
df.dm      = time.FRAME(df.dm)
df.pun     = time.FRAME(df.pun)
df.solar   = time.FRAME(df.solar)
df.wind    = time.FRAME(df.wind)
df.solar.AT = time.FRAME(df.solar.AT)
df.wind.AT = time.FRAME(df.wind.AT)
```

```
################################################################################
####   3.  CONVERT DATAFRAMES TO XTS FORMAT ###############################
################################################################################


xts.dm = xts(df.dm[,-1], order.by = df.dm$TIME)
xts.solar = xts(df.solar[,-1], order.by= df.solar$TIME)



################################################################################
####   4.  INTERPOLATE MISSING VALUES #####################################
################################################################################


################################################################################
####   4a. ONLY SOLAR: REPLACE NIGHTTIME NA's WITH 0
################################################################################


################################################################################
####   DEFINE SUBROUTINES  ##################################################

Sunrise.DE = function(Date){
  # Gives Time of Sunrise for "Date" in a matrix of size 1 * 2

  Location = c(13.413215, 52.521918) ##Coordinates of Berlin (Length, Width)
  Sunrise = sunriset(matrix(Location, nrow=1),
                     Date ,
                     direction="sunrise",
                     POSIXct.out = TRUE)
}

Sunset.DE = function(Date){
  # Gives Time of Sunset for for "Date" in a matrix of size 1 * 2

  Location = c( 13.413215, 52.521918) ##Coordinates of Berlin (Length, Width)
  Sunset = sunriset( matrix(Location, nrow=1),
                     Date ,
                     direction="sunset",
                     POSIXct.out = TRUE)
}


################################################################################
####   APPLY SUBROUTINES  ###################################################

for (TSO in names(df.solar[-1])){
  # Repeats the following procedure over the 4 colums
  # FZHertz", "Amprion", "TenneT.TSO", "Transnet.BW"

  if (length(subset(df.solar, is.na(df.solar[,TSO]) == TRUE)[,TSO])>0) {
    # Tests whether the column has no NA's
    # as this would lead to an error in the loop
```

```
    Missings = subset(df.solar, is.na(df.solar[,TSO]))[,c("TIME", TSO)]
    # Selects only the rows of df.solar that contain NA's in the column of
    # the index "TSO"
    # Was implemented for efficiency, as it would take a significant amount
    # of time to repeat this procedure over all rows

    sunrise.DE.List = lapply(Missings$TIME, Sunrise.DE)
    # Applies the above defined function (Sunrise.DE) on the dates of the
    # missing values for the column "TSO"
    # This will give a list containing two elements per calculated function
    # ( 1 per row of x.missing)

    sunrise.DE.df = do.call(rbind,sunrise.DE.List)
    # Transforms the List into a data.frame

    sunset.DE.List = lapply(Missings$TIME, Sunset.DE)
    # Applies the above defined function (Sunset.DE)on the dates of the
    # missing values for the column "TSO"
    # This will give a list containing two elements per calculated
    # function ( 1 per row of x.missing)

    sunset.DE.df = do.call(rbind,sunset.DE.List)
    # Transforms the List into a data.frame (easier to handle)

    Missings[Missings$TIME < sunrise.DE.df$time | Missings$TIME > sunset.DE.df$time,
        TSO] = as.numeric(0.0)
    # For all values of the df "Nissings", replace NA's by the value 0.0 whenever
    # it is before sunrise or after sunset

    xts.solar[Missings$TIME, TSO]=Missings[,TSO]
    # replaces missing solar values at night by 0 in the xts file containing
    # all solar data

  }
}


################################################################################
####   4b. NEAREST NEIGHBOUR INTERPOLATION
################################################################################


# Only applies for up to 4 consecutive NA's.
xts.solar = na.approx(xts.solar,na.rm=FALSE, maxgap=4)
xts.dm = na.approx(xts.dm ,na.rm=FALSE, maxgap=4)

################################################################################
####   4c. TREND INTERPOLATION ############################################
################################################################################

####   For values with strong seasonal variations within the day,
####   the week and the year (solar production and demand) we
####   interpolate by averaging between neighbouring days.


################################################################################
####   INTERPOLATE SOLAR BY DAILY TREND ##################################
```

```

# Build vector of quater-hours.
HM = function(Date){
  format(Date, "%H:%M")
}

Begin    = as.POSIXct("2011-03-31 00:00:00")
End      = as.POSIXct("2011-03-31 23:45:00")
quarters = seq(Begin, End, length.out=96)

quarters.day = HM(quarters)

# Interpolate SOLAR NA's by same hour values of neighbouring days.
solar.quarters.list = list()

for (quarter in quarters.day) {
# This loop splits up the orginal xts into xts files for every quarter hour.
# (One xts for all data for time "00:00", one for "00:15" etc...)
# This is done so that missing values can be interpolated on data for the
# same quarter hour.


  solar.quarters.list[[quarter]] = xts.solar[HM(index(xts.solar)) == quarter]
  #splits the solar data up into data by quarter-hours ( 1 xts per quarter)

  interpolated.values = na.approx(solar.quarters.list[[quarter]],
                              na.rm = FALSE,
                              maxgap = 2)
  # Interpolates NA's by Calculating the mean of the previous and the next
  # hour-quarterly solar generation value
  # ie: if there is the solar generation missing for "2011-03-31 08:00:00"
  # it averages the values for "2011-03-30 08:00:00"
  # and for "2011-04-01 08:00:00"


  xts.solar[index(solar.quarters.list[[quarter]]),] = interpolated.values
  #replaces the original data by the interpolated one.

  # print(summary(solar.quarters.list[[quarter]]))
  # print(summary(na.approx(solar.quarters.list[[quarter]],
  # na.rm=FALSE,
  # maxgap=4)))

}


################################################################################
####   INTERPOLATE DEMAND BY DAILY AND WEEKLY TREND #######################

# Build vector of days.
day.week = c(1,2,3,4,5,6,7)

#Define function that returns Weekday in format(0,1,2,3,4,5,6)
WeekDay = function(Date){
  as.POSIXlt(Date)$wday
}

#define list to store split up xts files
```

```
dm.day.list = list()

for (Day in day.week){
# This double loop splits up the orginal xts into xts files for every quarter
# hour and day of the week.(One xts for all data for time "00:00" on Monday,
# one for "00:00" on tuesday etc...)This is done so that missing values can be
# interpolated on data for the same time and the same day of the week.

  dm.day.list[[Day]]=list()
  #defining the nested list

  for (quarter in quarters.day ) {

    dm.day.list[[Day]][[quarter]]= xts.dm[HM(index(xts.dm)) == quarter &
                                         WeekDay(index(xts.dm)) == Day-1]
     # creates a new xts for each quarter hour for each day of the week


    interpolated.values = na.approx(dm.day.list[[Day]][[quarter]],
                                  na.rm=FALSE,
                                  maxgap=2)
    # for every such xts the NA's are interpolated.
    # ie: a NA on Monday 25th of March at 12:00 is interpolated by the demand
    # values on Monday the 18th of march at 12:00
    # and Monday the 1st of April at 12:00


    xts.dm[index(dm.day.list[[Day]][[quarter]])] = interpolated.values
    #all values of the original xts file get replaced by the interpolated ones
  }

}



###############################################################################
####   5. CONVERT BACK TO DATAFRAMES AND SAVE #############################
###############################################################################


dm.temp           = as.data.frame(xts.dm)
colnames(dm.temp) = "DEM"
df.dm$DEM         = dm.temp[ , 1]

solar.temp        = as.data.frame(xts.solar)
ColumnNamesSolar = colnames(df.solar)
df.solar          = cbind(df.solar[, 1], solar.temp)
rownames(df.solar) = c()
colnames(df.solar) = ColumnNamesSolar

# Save dataframes as '.Rdata' file for easy read-in in R.
save(df.pun, df.solar, df.solar.AT, df.wind, df.wind.AT, df.dm,
     file="MOEinterpolation/MOEdata_interp.Rdata"
)

# Save dataframes as '.csv' files for use with other software.
df.list    = list(df.pun, df.solar, df.solar.AT,
              df.wind, df.wind.AT, df.dm)
```

```

df.names  = c("df_pun.csv", "df_solar.csv", "df_solar_AT.csv",
                "df_wind.csv", "df_wind_AT.csv", "df_dm.csv")

lapply(1:length(df.list),
      function(i) write.csv((df.list[i]),
                           file = paste0("MOEinterpolation/MOEdata_interp_csv/",
                                         df.names[i])))

################################################################################
####   6. CLEAN UP ENVIRONMENT  ############################################
################################################################################


rm(list=ls()[! ls() %in% c("df.pun",
                          "df.solar",
                          "df.solar.AT",
                          "df.wind",
                          "df.wind.AT",
                          "df.dm"
)])
```

A.2.5. MOEmergedata.R

```

################################################################################
####   MOEmergedata.R  #####################################################
################################################################################
#
# Loads cleaned data set from 'MOErawdata'. Matches the timeframes and
# aggregates hourly into daily values.
#
# Input: 'MOEdata_interp.Rdata' file from the 'MOEinterpolation' Quantlet.
#
# Ouput: MOEdata_merge.csv   - data in table form
#        MOEdata_merge.Rdata - data in Rdata form
#
################################################################################

# Clear all variables.
rm(list = ls(all = TRUE))
graphics.off()

# Install and load libraries.
libraries = c("lubridate")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
    install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)



################################################################################
####   0.  SET WORKING DIRECTORY  ##########################################
################################################################################
####   ATTENTION: Working directory is assumed to be the root of the MOE
####   repository, not the MOEmergedata Quantlet subdirectory!!!

```

```
# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")



################################################################################
####   1. LOAD ENERGY MARKET DATA ##########################################
################################################################################


# Grab data from 'MOEinterpolation' Quantlet
load("MOEinterpolation/MOEdata_interp.Rdata")



################################################################################
####   2. CALCULATE DAILY VALUES  ##########################################
################################################################################


################################################################################
####   DEFINE SUBROUTINES  ###################################################

hour.MW.comp = function(x){
  # Computes MW per hour from 15 minute intervals
  y = sum(x)*0.25
  return(y)
}


################################################################################
####   APPLY SUBROUTINES  ####################################################

# Calculate daily averages for pun (price) and daily sums for energy production
# variables. 'df.solar' and 'df.wind' are quater-hourly data, so we have to
# multiply result by 0.25 to go from MW to MW/h (defined in hour.MW.comp()).
df.pun     = aggregate(list("PUN" = df.pun$PUN),
                       list("TIME" = cut(df.pun$TIME, "1 day")),
                       FUN = mean) # 'FUN = mean' because PUN is a price.
df.dm      = aggregate(list("DM" = df.dm$DEM),
                       list("TIME" = cut(df.dm$TIME, "1 day")),
                       FUN = sum)
df.solar   = aggregate(list(df.solar$`FzHertz`, df.solar$`Amprion`,
                         df.solar$`TenneT.TSO`, df.solar$`Transnet.BW`),
                       list("TIME" = cut(df.solar$TIME, "1 day")),
                       FUN = hour.MW.comp) # Note use of hour.MW.comp!
df.wind    = aggregate(list(df.wind$`FzHertz`, df.wind$`Amprion`,
                            df.wind$`TenneT.TSO`, df.wind$`Transnet.BW`),
                       list("TIME" = cut(df.wind$TIME, "day")),
                       FUN = hour.MW.comp) # Note use of hour.MW.comp!
df.solar.AT = aggregate(list("SOLAR.MW.AT" = df.solar.AT$`SOLAR.MW.AT`),
                       list("TIME" = cut(df.solar.AT$TIME, "1 day")),
                       FUN = sum)
df.wind.AT = aggregate(list("WIND.MW.AT" = df.wind.AT$`WIND.MW.AT`),
                       list("TIME" = cut(df.wind.AT$TIME, "1 day")),
                       FUN = sum)

```

```r
# Formating is lost when using aggregate(). Fix formating.
names(df.pun)       = c("TIME", "PUN")
names(df.dm)        = c("TIME", "DEM")
names(df.solar)     = c("TIME", "FzHertz", "Amprion", "TenneT.TSO",
                        "Transnet.BW")
names(df.wind)      = c("TIME", "FzHertz", "Amprion", "TenneT.TSO",
                        "Transnet.BW")
names(df.solar.AT)  = c("TIME", "SOLAR")
names(df.wind.AT)   = c("TIME", "WIND")
df.pun$TIME         = ymd(df.pun$TIME)
df.dm$TIME          = ymd(df.dm$TIME)
df.solar$TIME       = ymd(df.solar$TIME)
df.wind$TIME        = ymd(df.wind$TIME)
df.solar.AT$TIME    = ymd(df.solar.AT$TIME)
df.wind.AT$TIME     = ymd(df.wind.AT$TIME)



###############################################################################
####   3.  AGGREGATE 'DE' AND 'AT' DATA ###################################
###############################################################################


# Aggregate over the four DE producers of SOLAR and WIND.
df.solar = data.frame(df.solar$TIME, rowSums(df.solar[ ,-1]))
df.wind  = data.frame(df.wind$TIME, rowSums(df.wind[ ,-1]))

# Name the merged column.
names(df.solar) = c("TIME", "SOLAR")
names(df.wind)  = c("TIME", "WIND")

# Merge 'DE' and 'AT' renewable data.
df.solar[,-1]  = df.solar[,-1] + df.solar.AT[,-1]
df.wind[,-1]   = df.wind[,-1] + df.wind.AT[,-1]



###############################################################################
####   4.  BIND AND SAVE DATAFRAME ########################################
###############################################################################


# Bind dataframes.
df = cbind(df.pun, df.dm, df.solar, df.wind)
df = df[ -c(3,5,7) ] # Remove duplicate TIME columns

# Save dataframe as '.Rdata' file for easy read-in in R.
save(df, file="MOEmergedata/MOEdata_merge.Rdata")

# Save dataframe as '.csv' files for use with other software.
write.csv(df, file="MOEmergedata/MOEdata_merge.csv")



###############################################################################
####   5. CLEAN UP ENVIRONMENT ############################################
###############################################################################

```

```

# Remove everything except for "df" from environment.
rm(list=ls()[! ls() %in% c("df")])
```

A.2.6. MOEexploratory.R

```
################################################################################
####   MOEexploratory.R  ####################################################
################################################################################
#
# Creates exploratory plots of energy market variables.
#
# Input: '.Rdata' file from the 'MOEmergedata' and 'MOErawdata' Quantlet.
#
# Output: MOEplot_expl.tex     - plot in .tex format
#         MOEplot_expl.pdf     - plot in .pdf format
#         MOEplot_corr.tex     - plot in .tex format
#         MOEplot_corr.pdf     - plot in .pdf format
#         MOEtrend_solar.tex   - plot in .tex format
#         MOEtrend_solar.pdf   - plot in .pdf format
#         MOEtrend_demand.tex  - plot in .tex format
#         MOEtrend_demand.pdf  - plot in .pdf format
#
################################################################################

# Clear all variables.
rm(list = ls(all = TRUE))
graphics.off()

# Install and load libraries.
libraries = c("tidyr", "ggplot2", "tikzDevice", "cowplot", "scales")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
  install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)


################################################################################
####   0.  SET WORKING DIRECTORY  ##########################################
################################################################################

####   ATTENTION: Working directory is assumed to be the root of the MOE
####   repository, not the MOEmergedata Quantlet subdirectory!!!


# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")


################################################################################
####   1.  LOAD DATA   ######################################################
################################################################################

# Grab data from 'MOEmergedata' and 'MOErawdata' Quantlet
load("MOErawdata/MOEdata_clean.Rdata")
load("MOEmergedata/MOEdata_merge.Rdata")

# Change variable names for better representation
```

```r
names(df) <- c("TIME", "PRICE", "DEMAND", "SOLAR", "WIND")

# Create tidy datasets
tidy.df = df %>% gather(key = VAR, value = ENERGY, 3:ncol(df))
tidy.df = tidy.df %>% gather(key = LAB, value = PRICE, 2)


################################################################################
####   2.  CREATE PLOTS    ##################################################
################################################################################


################################################################################
####   2.1 EXPLORATORY PLOT  ################################################

# Create ENERGY over TIME plot
plot_pwr = ggplot(data=tidy.df, aes(x = TIME, y = ENERGY)) +
    geom_point(size = 0.5) +
    ggtitle(label = "The German and Austrian Energy Market",
            subtitle = "Selected Day-Ahead Variables, 2015--2017") +
    xlab(label="") +
    ylab(label = "Energy in MWh") +
    scale_y_continuous(label = comma) +
    theme_bw() +
    facet_grid(VAR ~ ., scales = "free")


# Create PRICE over TIME plot
plot_pun = ggplot(data=tidy.df, aes(x = TIME, y = PRICE)) +
    geom_point(size = 0.5) +
    labs(x = "", y = "Price in Euro/MWh") +
    scale_y_continuous(label = comma) +
    theme_bw() +
    facet_grid(LAB ~ ., scales = "free")


# Bind plots together
plot_exp = plot_grid(plot_pwr, plot_pun, align = "v", nrow = 2,
                     rel_heights = c(1, 0.39)
                     )


################################################################################
####   2.2 CORRELATION PLOT  ################################################

# Create PRICE over ENERGY plot
plot_pr_rn = ggplot(data= tidy.df[tidy.df$VAR != "DEMAND", ],
                    aes(y = PRICE, x = ENERGY)) +
        geom_point(size=0.5) +
        geom_smooth(method="lm", aes(fill= TIME), fullrange = T) +
        ggtitle(label = "Co-movements on the Energy Market 2015--2017",
                subtitle = "Selected Day-Ahead Variables") +
        xlab(label = "") +
        ylab(label = "Price in Euro/MWh") +
        facet_grid(VAR ~., scales = "free") +
        theme_bw()

```

```
# Create PRICE over DEMAND plot
plot_pr_de = ggplot(data= tidy.df[tidy.df$VAR == "DEMAND", ],
                  aes(y = PRICE, x = ENERGY)) +
       geom_point(size=0.5) +
       geom_smooth(method="lm", aes(fill=‘TIME‘),fullrange = T) +
       xlab(label = "Energy in MWh") +
       ylab(label = "Price in Euro/MWh") +
       theme_bw() +
       facet_grid(VAR ~., scales = "free")


# Bind plots together
plot_corr = plot_grid(plot_pr_rn, plot_pr_de, align = "v", nrow = 2,
                                        rel_heights = c(0.7, 0.4))


##############################################################################
####   2.3 SOLAR TREND  #####################################################

trend_solar = ggplot(data=df.solar, aes(x = TIME, y = FzHertz)) +
    geom_point(size = 0.5) +
    labs(x = "", y = "Production in MWh") +
    ggtitle(label = "Example of Daily Pattern in Solar Energy Generation") +
    scale_x_datetime(limits = c(as.POSIXct('2016-07-11 23:30:00'),
                                as.POSIXct('2016-07-13 00:30:00')),
                     labels = date_format("%H:%M"),
                     breaks = date_breaks("3 hour"),
                     expand = c(0,0)) +
    scale_y_continuous(label = comma) +
    theme_bw()


##############################################################################
####   2.4  DEMAND TREND   ##################################################

trend_demand = ggplot(data=df.dm, aes(x = TIME, y = DEM)) +
    geom_point(size = 0.5) +
    labs(x = "", y = "Demand in MWh") +
    ggtitle(label = "Example of Weekly Pattern in Energy Demand") +
    scale_x_datetime(limits = c(as.POSIXct('2016-07-11 00:00:00'),
                                as.POSIXct('2016-08-01 23:45:00')),
                     labels = date_format("%A"),
                     breaks = date_breaks("7 day"),
                     expand = c(0,0)) +
    scale_y_continuous(label = comma) +
    theme_bw()


##############################################################################
####   3.  SAVE PLOTS AS TEX FILE   #########################################
##############################################################################

# Save explorative plot as .tex file
tikz(file = "MOEexploratory/MOEplot_expl.tex", width = 6, height = 8)
plot(plot_exp)
dev.off()

# Save correlation plot as .tex file
```

```
tikz(file = "MOEexploratory/MOEplot_corr.tex", width = 6, height = 8)
plot(plot_corr)
dev.off()

# Save solar trend plot as .tex file
tikz(file = "MOEexploratory/MOEtrend_solar.tex", width = 7, height = 3)
plot(trend_solar)
dev.off()

# Save demand trend plot as .tex file
tikz(file = "MOEexploratory/MOEtrend_demand.tex", width = 7, height = 3)
plot(trend_demand)
dev.off()

# Save explorative plot as .pdf file
pdf("MOEexploratory/MOEplot_expl.pdf")
plot(plot_exp)
dev.off()

# Save correlation plot as .pdf file
pdf("MOEexploratory/MOEplot_corr.pdf")
plot(plot_corr)
dev.off()

# Save solar_trend plot as .pdf file
pdf("MOEexploratory/MOEtrend_solar.pdf", width = 7, height = 3)
plot(trend_solar)
dev.off()

# Save demand trend plot as .pdf file
pdf("MOEexploratory/MOEtrend_demand.pdf", width = 7, height = 3)
plot(trend_demand)
dev.off()




################################################################################
####   4. CLEAN UP ENVIRONMENT #############################################
################################################################################

rm(list=ls()[! ls() %in% c("df",
                           "plot_exp",
                           "plot_corr",
                           "trend_solar",
                           "trend_demand"
                           )])
```

A.2.7. MOEtimedummies.R

```
################################################################################
####   MOEtimedummies.R  #####################################################
################################################################################
#
# This code defines the function "YMDDummy" that adds a matrix of dummy
# variables for years, months and days on the right side of an xts object.
# The parameter "FullDat.xts" specifies the xts object for which to generate
# time dummies
```

```
# For each type of dummy (Years, Months, Days), a dummy to be left out can be
# specified in the 3 other parameters.
# e.g: the parameter del.Y specifies the number of the year to be left out
# (1 for the oldest year, 2 for the second oldest year etc...)
# e.g: the parameter del.M specifies the number of the Month to be left out
# (1 for January, 2 for february...)
# e.g: the parameter del.D specifies the number of the Day to be left out
# (1 for Sunday, 2 for Monday, 3 for Tuesday etc.. until 7 for Saturday)

################################################################################


# Install and load libraries.
libraries = c("xts")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
  install.packages(x)
})
lapply(libraries, library, quietly = TRUE, character.only = TRUE)



################################################################################
####   1. DEFINE YMDdummy FUNCTION ########################################
################################################################################

YMDDummy = function(FullDat.xts, del.Y = 1, del.M = 1, del.D = 1){

  Year.min                   = format(min(index(FullDat.xts)), "%Y")
  Year.max                   = format(max(index(FullDat.xts)), "%Y")
  Year.Vector                = seq(from=Year.min, to=Year.max, by=1)
  Year.min.number            = as.numeric(Year.min)
  Year.max.number            = as.numeric(Year.max)

  ##############################################################################
  ####   Step 0: Check parameters ###########################################

  if(is(FullDat.xts, "xts")==FALSE){ ###TO BE FIXED
      stop("The called object is not an xts")
  }else if(del.Y<1 | del.Y> Year.max.number-Year.min.number+1 ){
      stop('The specified Year to be left away in the dummy variables does not exist')
  }else if(del.M<1 | del.M>12){
      stop('The specified Month to be left away in the dummy variables does not exist')
  }else if(del.D<1| del.D>7){
      stop('The specified Month to be left away in the dummy variables does not exist')
  }else{


  ##############################################################################
  ####   Step 1: Create the dummy variables for years, months and days ####

  Length = length(FullDat.xts[,1])

  ReturnYear = function(x){
    #Gives the year for a specific date
    format(index(x), "%Y")
  }

  ReturnMonth = function(x){
```

```
  #Gives the Month for a specific date
  format(index(x), "%m")
}

ReturnDay = function(x){
  #Gives the year for a specific date
  as.POSIXlt(index(x))$wday
}


#############################################################################
####      1.1 Dummy variables for Years #################################

NumYears                   = Year.max.number-Year.min.number+1
Year.Dummy.matrix          = matrix(,nrow = Length , ncol=NumYears)
colnames(Year.Dummy.matrix) = Year.Vector

for (i in 1:length(Year.Vector)) {
    Year.Dummy.matrix[,i] = ReturnYear(FullDat.xts) == Year.Vector[i]
}


#############################################################################
####      1.2 Dummy variables for Months ################################

Month = c("01", "02", "03", "04", "05", "06",
         "07", "08", "09", "10", "11", "12")
Month.Dummy.matrix = matrix(,nrow = Length, ncol=length(Month))
colnames(Month.Dummy.matrix) = Month

for (i in 1:length(Month)) {
  Month.Dummy.matrix[,i] = ReturnMonth(FullDat.xts) == Month[i]
}


#############################################################################
####      1.3 Dummy variables for Days  #################################

n=6
days.of.the.week        = c("Sunday",
                            "Monday",
                            "Tuesday",
                            "Wednesday",
                            "Thursday",
                            "Friday",
                            "Saturday")

Day.Dummy.matrix        = matrix(,nrow = Length, ncol=n+1)
colnames(Day.Dummy.matrix) = days.of.the.week
Weekdays                = ReturnDay(FullDat.xts)
# vector of the week day of each day
# ( in numbers : 0=Sunday, 1=Monday...6=Saturday)

for (i in 0:n) {
    Day.Dummy.matrix[,i+1] = Weekdays== i
}
```

```
  ################################################################################
  ####   Step 2: Cbind the dummy matrices ################################

  Time.Dummy.matrix      = cbind(Year.Dummy.matrix[,-del.Y],
                                 Month.Dummy.matrix[,-del.M],
                                 Day.Dummy.matrix[,-del.D])
  Time.Dummy.matrix.xts   = xts(x=Time.Dummy.matrix, order.by=index(FullDat.xts))
  Time.Dummy.matrix.xts.num = Time.Dummy.matrix.xts+0 # Converts the data to nummeric

  ################################################################################
  ####   Step 3 Cbind the dummy matrices #################################

  FullDat.xts = cbind(FullDat.xts, Time.Dummy.matrix.xts.num)

  }
}
```

A.2.8. MOEregression.R

```

################################################################################
####   MOEregression.R  ####################################################
################################################################################
#
# Performs an OLS and the Prais-Winsten regression as well as the following
# tests:
#
#   - Augmented Dickey-Fuller test and Philipps-Perron test for stationarity of
#     the data (with trend and constant)
#   - Durbin-Watson test for autocorrelation in the OLS
#   - Breusch-Pagan for heteroscedasticity in the OLS
#   - Durbin-Watson test for autocorrelation in the Prais-Winsten regression
#
# The ACF and PACF are plotted for OLS and Prais-Winsten as well.
#
# Input: 'MOEdata_interp.Rdata' from the 'MOEinterpolation' Quantlet.
#
# Ouput: 'PACF_OLS.jpg'       - ACF and PACF plot for OLS
#        'PACF_PraisWinsten.jpg' - ACF and PACF plot for Prais-Winsten
#        'ADFandPPTEST'        - LaTeX table for ADF and PP tests
#        'OLS_DWandBPTEST'     - LaTeX table for Durbini-Watson
#                               and Breusch-Pagan tests on the OLS
#        'PraisWinstenRegCoefficients' - LaTeX table of PW reg. coefficents
#        'PraisWinstenGoFDW'   - LaTeX table of R^2, Adj. R^2 and DW test
#                               results of the Prais-Winsten regression
#
################################################################################


# Clear all variables.
rm(list = ls(all = TRUE))
graphics.off()

# Install and load libraries.
libraries = c("xts", "prais","lmtest", "forecast", "astsa", "tseries", "xtable")
lapply(libraries, function(x) if (!(x %in% installed.packages())) {
  install.packages(x)
})
```

```r
lapply(libraries, library, quietly = TRUE, character.only = TRUE)



###############################################################################
####   0.  SET WORKING DIRECTORY ##########################################
###############################################################################

####   ATTENTION: Working directory is assumed to be the root of the MOE
####   repository, not the MOEmergedata Quantlet subdirectory!!!


# If needed, set working directory accordingly:
#setwd("path/to/MOE_repository")



###############################################################################
####   1.  LOAD CLEAN DATA  ################################################
###############################################################################

load("MOEmergedata/MOEdata_merge.Rdata")



###############################################################################
####   2.  PREPARE DATA  ###################################################
###############################################################################


###############################################################################
####   INTERPOLATE WIND DATA ON DAILY BASIS ##############################

xts.mydata        = xts(df[, -1], order.by=df$TIME)
xts.mydata[,"WIND"] = na.approx(xts.mydata[,"WIND"], na.rm=TRUE, maxgap=3)


###############################################################################
####   ADD TIME DUMMIES TO XTS DATAFRAME ##################################

source("MOEtimedummies/MOEtimedummies.R")

xts.mydata = YMDDummy(xts.mydata)


###############################################################################
####   TRANSFORM TO TS DATAFRAME AND ADJUST UNITS #########################

ts.mydata = as.ts(xts.mydata)

# changes the scale of the data to facilitate the interpretation
# generation data is now expressed in mean hourly generation in GWh(thus GW)
# Price is still in EUR/MWh
ts.mydata[,2:4] = as.ts(xts.mydata)[,2:4]/24000



###############################################################################
```

```
####   3. PERFORM TESTS AND REGRESSIONS ##################################
#######################################################################


#######################################################################
####  Part 1. Tests for Stationarity ######################################


#######################################################################
####      Augmented Dickey Fuller Test

Results.ADF = rep(NA, 4)

for (Column in 1:4) {
    # Performs the ADF test for the variables Price, Demand, Solar and Wind
    # the p-values are stored in Results.ADF
    # H0: non-stationary ( with time trend and constant)
    Results.ADF[Column]=tseries::adf.test(ts.mydata[, Column],
                                          alternative = "stationary")$p.value
}


#######################################################################
####   Philipps-Perron Test

Results.PP = rep(NA, 4)

for (Column in 1:4) {
    # Performs the PP test for the variables Price, Demand, Solar and Wind
    # the p-values are stored in Results.PP
    # H0: non-stationary ( with time trend and constant)
    Results.PP[Column]= tseries::pp.test(ts.mydata[, Column])$p.value
}


# Arrange results into LaTeX table
Table1 = data.frame(Results.ADF, Results.PP)
colnames(Table1) = c("A. Dickey Fuller", "Philipps-perron")
rownames(Table1) = c("El. Price","El. Demand", "Solar Gen.", "Wind Gen.")
TABLE1 = xtable(Table1)
print.xtable(TABLE1, type="latex", file="MOEregression/ADFandPPTEST.tex")


#######################################################################
####   Part 2. Basic OLS, Autocorrelation Structure #######################


#######################################################################
####      Basic OLS

OLS = tslm(PUN ~ ts.mydata[,-1], ts.mydata)


#######################################################################
####      Durbin-Watson Test
####      (check for autocorrelation of the disturbances)

DWTEST.OLS = dwtest(OLS)
```

```


################################################################################
####       Breusch-Pagan Ttest for Heteroscedasticity
####       (Under H0 the test statistic of the Breusch-Pagan test follows a
####       chi-squared distribution degrees of freedom)

BPTEST.OLS = bptest(OLS)


# Arrange results into LaTeX table
Table2 = data.frame(DWTEST.OLS$p.value,BPTEST.OLS$p.value)
colnames(Table2) = c("Durbin-Watson", "Breusch-Pagan")
row.names(Table2) = "p-value"
TABLE2 = xtable(Table2)
print.xtable(TABLE2, type="latex", file="MOEregression/OLS_DWandBPTEST.tex")


# Generate jpeg file with PACF and ACF of OLS
jpeg('MOEregression/PACF_OLS.jpg')
acf2(OLS$residuals, main="OLS residuals")
dev.off()


################################################################################
####    Part 3. Prais-Winsten Regression for modelling of AR(1) process ####


################################################################################
####       Prais -Winsten Generalised Least Squares Regression
####       (modelling the distrubances with an AR(1) process)

PWReg = prais.winsten(PUN ~ .,ts.mydata,iter = 50,rho = 0, tol = 1e-08)

# Generate a jpeg for ACF and PACF of PW regression
jpeg('MOEregression/PACF_PraisWinsten.jpg')
acf2(PWReg[[1]]$residuals, main="Prais-Winsten residuals.tex")
dev.off()

# Arrange results into LaTeX table
coef = PWReg[[1]][, drop=F]$coefficients
coef = rbind(coef[-1,], coef[1,])

rho = PWReg[[2]]
rho = as.numeric(c(PWReg[[2]][1,1], NA ,PWReg[[2]][1,2], NA))

Table3 = rbind(coef, rho)
Table3 = as.data.frame(Table3)
rownames(Table3) = c("Demand", "Solar Gen.", "Wind Gen.",
                     "Year 2016", "Year 2017", "February",
                     "March", "April", "May", "June", "July",
                     "August", "September", "October", "November",
                     "December", "Monday", "Tuesday", "Wednesday",
                     "Thursday","Friday", "Saturday", "Intercept",
                     "\rho (AR1)")

TABLE3 = xtable(Table3,
               caption = "Prais-Winsten regression results",
```

```
              align =c("l", "r","r", "r","r"),
              digits=2)

print.xtable(TABLE3, type ="latex",
            file = "MOEregression/PraisWinstenRegCoefficients.tex")


################################################################################
####     Durbin Watson Test
####     (for autocorrelation after correcting for serial correlation)

DWTEST.PW = dwtest(PWReg[[1]])

# generates a Latex table with the R^2, the Adj R^2 and the Result of the durbin
# watson test fopr the prais winsten regression
Table4 = data.frame(PWReg[[1]]$r.squared,
          PWReg[[1]]$adj.r.squared,
          DWTEST.PW$p.value)
colnames(Table4) = c("$R^2$","$Adj. R^2$","Durbin-Watson (p value )")
rownames(Table4) = ""


TABLE4 = xtable(Table4,
              caption = "Prais-Winsten regression results",
              align=c("l","c", "c", "c"),
              digits=2)

print.xtable(TABLE4,
            type="latex",
            file="MOEregression/PraisWinstenGoFDW.tex",
            sanitize.text.function=function(x){x})



################################################################################
####    5. CLEAN UP ENVIRONMENT ############################################
################################################################################


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

Declaration of Authorship

We hereby confirm that we have authored this Seminar paper independently and without use of others than the indicated sources. All passages which are literally or in general matter taken out of publications or other sources are marked as such.

Felix Germaine, Manuel Pfeuffer *and* Bruno Puri
Berlin, August 12, 2018

References

CLÒ, Stefano, Alessandra CATALDI, and Pietro ZOPPOLI (2015). "The merit-order effect in the Italian power market: The impact of solar and wind generation on national wholesale electricity prices". In: *Energy Policy* 77, pp. 79–88.