# R Applications for stationary and weakly dependent time series models

## Introductory Remarks

This file builds on the previous R Activities. If you have not yet gone through those files, please make sure to do so before tackling this R Activity.

## Static Time Series Models

As discussed in the lecture, a static time series model is of the form

$$y_t = \beta_0 + \beta_1 z_t + \epsilon_t$$

where $y_t$ is some outcome at time $t$ (e.g. inflation at time $t$) and $z_t$ is the regressor of interest at time $t$ (e.g. unemployment at time $t$). It turns out that estimating this equation in R is done with the usual "lm" command *as long as all assumptions you discussed in class are satisfied*. Since you already know how to do this, we will skip this and focus on new content here. Note that static time series models rarely satisfy all the assumptions you need to be able to run OLS on the above equation. Thus, when working with time series data, it is often better to steer away from such models.

## Defining and Plotting Time Series Data in R

When working with time series data in R, it is useful to indicate to ("tell") R that we are working with time series data.[1] We do this because once R knows that we are working with time series data, we can easily access a whole range of commands to, say, lag certain variables.

Unfortunately, R makes this process slightly more complicated than it has to be. Specifically, we need to differentiate between equispaced and irregular time series data. Equispaced data are collected at regular points in time (e.g. yearly or quarterly), while irregular data are collected, well, at irregular points in time. An example of an irregular variable are daily financial data which are not recorded on weekends. The point to keep in mind is that if you're working with irregular data, you will need a time or date variable to indicate the precise time of each observation. When working with equispaced data, this is not necessary.

As an example, consider loading a dataset with *GDP per capita* data for the UK from 2010 - 2020. If the data you are loading is just one variable with 11 observations of *GDP per capita*, then you are working with equispaced data. In this case, all you need to do is to tell R when this data starts and R will do the rest for you. In other words, by telling R that the first observation of *GDP per capita* is from 2010 and that you're working with yearly data, R will automatically associate the second observation with 2011, the third with 2012, and so on. Now suppose that the *GDP per capita* series you're loading into R contains only 8 observations (instead of 11). Then, you don't know which particular year these 8 observations belong to. In such a case, you would need to also have a *year* variable, which would assign each of these 8 observations to a particular year. In this case, you're working with irregular data.

### Equispaced Time Series Data

The simplest function to create a time series object in R is the "ts" function (see "?ts" for help). To see this function in action, consider loading the BARIUM data into R by typing

[1] In case you are familiar with Stata, "telling R" that we are working with time series data is similar to the "tsset" command in Stata.

```
data("barium") # loading the barium data
```

This dataset contains monthly data on imports of barium chloride from China between February 1978 and December 1988. Note that when you look at the data, you don't actually see a date variable to indicate the timespan when we observe the data. The variable indicating the imports is called *chnimp*. Given that we don't have a date variable, if this data was irregular we wouldn't know what observation of *chnimp* is associated with date. But given that the data is equispaced, we know that the first observation is February 1978, the second is March 1978, and so on. We can turn this variable into a ts object by typing

```
imports<- ts(barium$chnimp, start=c(1978,2), frequency=12)
```

The first argument in the ts function tells R what object we want to turn into a ts object. Note that we are here only turning the *chnimp* variable into a ts object. In general, you can also put in the names of full datasets, as well will see below. The second argument tells R the time of the first observation. In our case, this is February 1978. Lastly, we need to tell R the frequency, i.e. the number of observations per unit of time. In our case, we have monthly data, which implies that we have 12 observations per year. If we had quarterly data, we would write "frequency = 4" (then, "start=c(1978,2)" would imply the second quarter of 1978).

Once we've told R that these Chinese imports of barium chloride are a time series object, we can plot this variable very easily by typing

```
plot(imports)
```

The resulting figure should appear in the bottom right quadrant of RStudio. As you can see, the y-axis denotes the imports and the x-axis the time. Note that the x-axis is automatically formatted properly (even though there is no date variable in the dataset!). This is because R knows that this variable is a time series object.

### Irregular Time Series Data

For our purposes, the "ts" command discussed above will be enough. However, since many datasets will have irregular time series observations, it is useful to understand how to load this data.

Two of the most important packages to time-set irregular time series data in R are "xts" and "zoo" (see ?xts and ?zoo for help). We will look at how "zoo" works, which can also be used for equispaced time series.

The main point to keep in mind with irregular time series is that since the data are not necessarily equispaced, each observation of an irregular time series variable needs to be associated with a time or a date in another variable. This is almost always the case, and hence not a big requirement. Let's load the INTDEF data, a dataset containing interest rates, into R as we always do by typing

```
data("intdef")
```

Notice that there is a variable *year* and recall that in the BARIUM data from above, there was no time/date variable. To time set this dataset, we can type (make sure you have installed and loaded the zoo package)

```
intdef_ts <- zoo(intdef, order.by=intdef$year)
```

The first entry, in our case, is the full dataset we loaded. Note that it could also be a specific variable. The second input into the zoo function is the time variable. In our case this is the year variable. Notice that the zoo command is very simple. Indeed, since most data you will encounter will contain a variable with the time (e.g. year, quarter, or day), it might be good to get used to always using the zoo command. Finally, to plot the interest rate over time, we can use the same command as above, i.e.

```
plot(intdef_ts$i3)
```

Again, notice that we don't need to specify the x-axis. R does this automatically. To really clarify this point, consider what happens if you just plot the variable *i3* withtout telling R that it is a time series object, i.e. type

```
plot(intdef$i3, type="l")
```

Apart from the fact that we need to specify the type of the plot to be a line plot (otherwise we get a scatter plot), the x-axis of this plot is just an index from 1 to something above 50. In order to manually get the years on the x-axis, we would have to type

```
plot(intdef$year, intdef$i3, type="l")
```

We now have manually created the same plot that we were able to create in a simpler way by just telling R that this is a time series object. Specifically, by telling R that *i3* is a time series object, R automatically formatted the x-axis of the plot. When not giving this information to R, we had to manually tell R what we want on the x-axis. While in this particular example this time-setting may not seem incredibly useful, it is very useful when lagging variables and when doing other time series related things.

## Finite Distributed Lag (FDL) Models

A FDL model with $q$ lags has the following form

$$y_t = \alpha_0 + \gamma_0 z_t + \gamma_1 z_{t-1} + ... + \gamma_q z_{t-q} + \epsilon_t$$

When attempting to estimate such a model in R, we make use of the "dynlm" function (see ?dynlm for help). Intuitively, you can think of "dynlm" as the time series equivalent of the "lm" function that we have used extensively so far. To use this function, you need to install the "dynlm" package and then load it.

To fix ideas, let's load the FERTIL3 dataset into R and time-set it

```
data("fertil3")
fertil3_ts <- ts(fertil3, start=1913)
```

Note two things. First, we omit the "frequency" option in the "ts" command as we have yearly data (try adding "frequency=1" and check for yourself that you get the same ts object). Second, if you look at the data, you can see a *year* variable. Therefore, we could equivalently use the zoo command to time set the data by typing

```
fertil3_ts2 <- zoo(fertil3, order.by = fertil3$year)
```

This data, which you have seen already in the lectures, contains yearly information on the general fertility rate ( *gfr*) and the personal tax exemption ( *pe*) for the years 1913 through 1984. Similar to your class, we will use the dummy *ww2* as an indicator for the years when WW2 took place. The dummy *pill* indicates the availability of birth control. Our aim here will be to run the following regression

$$gfr_t = \alpha_0 + \gamma_0 pe_t + \gamma_1 pe_{t-1} + \gamma_2 pe_{t-2} + \delta_1 ww2_t + \delta_2 pill_t + \epsilon_t$$

In words, we are regressing the fertility rate in year t on personal tax exemptions in year t plus two lags, while controlling for the dummies *ww2* and *pill*. To run this regression using the "dynlm" function, type

```
fert_reg <- dynlm(gfr ~ pe + L(pe) + L(pe, 2) + ww2 + pill, data=fertil3_ts)
summary(fert_reg)
```

Note that the overall use of this function is very similar to the use of the "lm" function. The first input is the formula, and the second is the dataset we are using. As an exercise, you can try to specify the dataset created when time-setting the data using the "zoo" function to see that the results don't change.

The only new thing in the above are the "L(pe)" and "L(pe, 2)." By looking at the estimating equation, it is clear that $pe = pe_t$, $L(pe) = pe_{t-1}$, and $L(pe, 2) = pe_{t-2}$. In words, L is the lag operator. "L(pe)" will lag the variable *pe* once, "L(pe, 2)" will lag it twice, "L(pe, 3)" would lag it three times, and so forth. We are able to use this lag operator because we have time-set our data. Otherwise, this would not work.

In the lecture, you learned that the long-run propensity (LRP) of the FDL model is given by the sum of the coefficients on $z$. In our example, this is equivalent to

$$LRP = \gamma_0 + \gamma_1 + \gamma_2$$

.

To test the null hypothesis whether the LRP is equal to zero, we could run the by now familiar command "linearHypothesis"

```
linearHypothesis(fert_reg, "pe + L(pe) + L(pe, 2) = 0")
```

We get an F-statistic of 11.421 and an associated p-value of 0.001241. We therefore clearly reject the null hypothesis that the LRP is equal to zero at the 5% significance level.

## AR(p) Models

It turns out that the "dynlm" function can also be used to run AR(p) models. To be clear, an AR(p) models takes the following form

$$y_t = \beta_0 + \beta_1 y_{t-1} + ... + \beta_p y_{t-p} + \epsilon_t$$

To see this applied in R, load the NYSE dataset, a dataset containing weekly stock returns, and time-set it

```
data("nyse") # loading the nyse data
nyse_ts <- zoo(nyse, order.by = nyse$t)
```

Let's regress stock returns on their own lags. We'll start by including one lag, and progressively increase to three lags. In R,

```
reg_ar1 <- dynlm(return ~ L(return), data = nyse_ts) # AR(1) Model
reg_ar2 <- dynlm(return ~ L(return) + L(return, 2), data = nyse_ts) # AR(2) Model
reg_ar3 <- dynlm(return ~ L(return) + L(return, 2) +
                   L(return, 3), data = nyse_ts) # AR(3) Model
summary(reg_ar1)
summary(reg_ar2)
summary(reg_ar3)
```

As you can see neither of the lags is significant. As an exercise you can try to test for the joint significance of the three lags. Most importantly, however, you have seen that implementing an AR(p) model in R is very straightforward.

## Testing for Serial Correlation

One big potential issue when working with time series data is the dependence of the data. In other words, we might be worried that the errors of our model are serially correlated over time. To see this mathematically, consider the following FDL model

$$inf_t = \beta_0 + \beta_1 unemp_t + u_t$$

where $inf_t$ is inflation at time $t$ and $unemp_t$ is unemployment at time $t$. Serial correlation here would imply that the errors $u_t$ are correlated over time. An assumption that is often made is that this serial correlation follows an AR(p) process, i.e.

$$u_t = \alpha_0 + \alpha_1 u_{t-1} + ... + \alpha_p u_{t-p} + v_t$$

with $v_t$ being white noise. The most common assumption is that these errors follow an AR(1) process. Note that if $\alpha_1 = \alpha_2 = ... = \alpha_p = 0$, then the errors are not serially correlated.

A manual way of testing for serial correlation is to run our equation of interest using the "dynlm" command, and then to calculate the residuals $\hat{u}_t$. We can then regress the residuals $\hat{u}_t$ on their lagged values (e.g. $\hat{u_{t-1}}$, $\hat{u_{t-2}}$, etc.). You can think of this regression using the residuals as the best approximation we can make to the AR(p) model written just above (since we never observe the errors $u_t$, our best guess is to approximate

them with the residuals). Hence, testing whether the coefficients on the lagged residuals are jointly zero is equivalent to testing for serial correlation. This procedure of testing whether the coefficients on the lagged residuals are jointly zero is often called the Breusch-Godfrey (BG) test. In class, you also learned about another (similar) test called the Durbin-Watson (DW) test.

Let's try to implement all three versions of testing for serial correlation in R using the PHILLIPS dataset.

```
data("phillips") # load data
phillips_ts <- zoo(phillips, order.by = phillips$year) # ts data
```

To run all three tests, run the following code

```
phil_reg <- dynlm(inf ~ unem, data=phillips_ts)
summary(phil_reg)
# test for autocorrelation manually
phil_res <- resid(phil_reg) # get residuals
resid_test <- dynlm(phil_res ~ L(phil_res)) # reg residuals on lagged residuals
summary(resid_test) # display results
# BG and DW test
bgtest(phil_reg) # BG test
dwtest(phil_reg) # DW test
```

The first two lines of code run the regression of inflation on unemployment and display the results. The next three lines of code (i) calculate the residuals from our regression, (ii) regress the residuals on their lag (assuming an AR(1) process for the errors), and (iii) summarize the results. As we can see, the coefficient on the lagged residuals is significant at the 1% level, thus suggesting that serial correlation is an issue in our regression. The last two lines of code implement the BG and DW tests. Note that for these two commands you need to have the "lmtest" package installed. The p-values we receive from both of these tests yield the same conclusion as our manual test.

Notice that the DW test is usually only used when we assume the errors follow an AR(1) process. The BG test, as well as the manual implementation, can both be used for higher order AR processes of the errors as well. To see this, type

```
bgtest(phil_reg, order=2) # BG test
```

By adding in the option "order," you can add more lags $p$ to your AR(p) process. Lastly, note that the BG test allows you to conduct either an F-test or an LM test that the lags are jointly zero. The default of the "bgtest" function is to use the LM test. If you want it to do the F-test, type the following

```
bgtest(phil_reg, order=2, type="F") # BG test w/ Ftest
```

## HAC Standard Errors

Given the results above, serial correlation is clearly an issue in our regressions. From lecture we know that serial correlation will invalidate our standard errors. Luckily, we also know that there is a solution to this problem, namely computing heteroskedasticity and autocorrelation consistent (HAC) standard errors.

In R, this turns out to be very easy to do. In what follows, we will make use of the "lmtest" and the "sandwich" package, so make sure both are installed. Recall our output from the regression of inflation on unemployment,

```
summary(phil_reg)
coeftest(phil_reg)
```

Note that the second line using the "coeftest" function displays the same as the first. The standard errors reported are calculated under the usual homoskedasticity assumption. To report HAC standard errors, type

```
coeftest(phil_reg, vcovHAC)
```

As you can see, in this example the standard errors change only slightly, and the significance levels of the coefficients do not change. Important for our purposes, calculating HAC standard errors in R is very easy.