B3_HWA2 Example

Lukas Arnroth

17 juni 2017

In this document you will be provided with an example that you can use for your own assignment. I will in large copy the settings of your assignment with a time series not part of your data. Lets begin by simulating the data. I will simulate a AR(1) time series setting $\phi_1 = 0.5$

```
set.seed(43770)
y <- arima.sim(model = list(ar = 0.5), n = 404)
```

I will move on assuming I only have the first 400 observations to use, and therefore split the y vector as follows

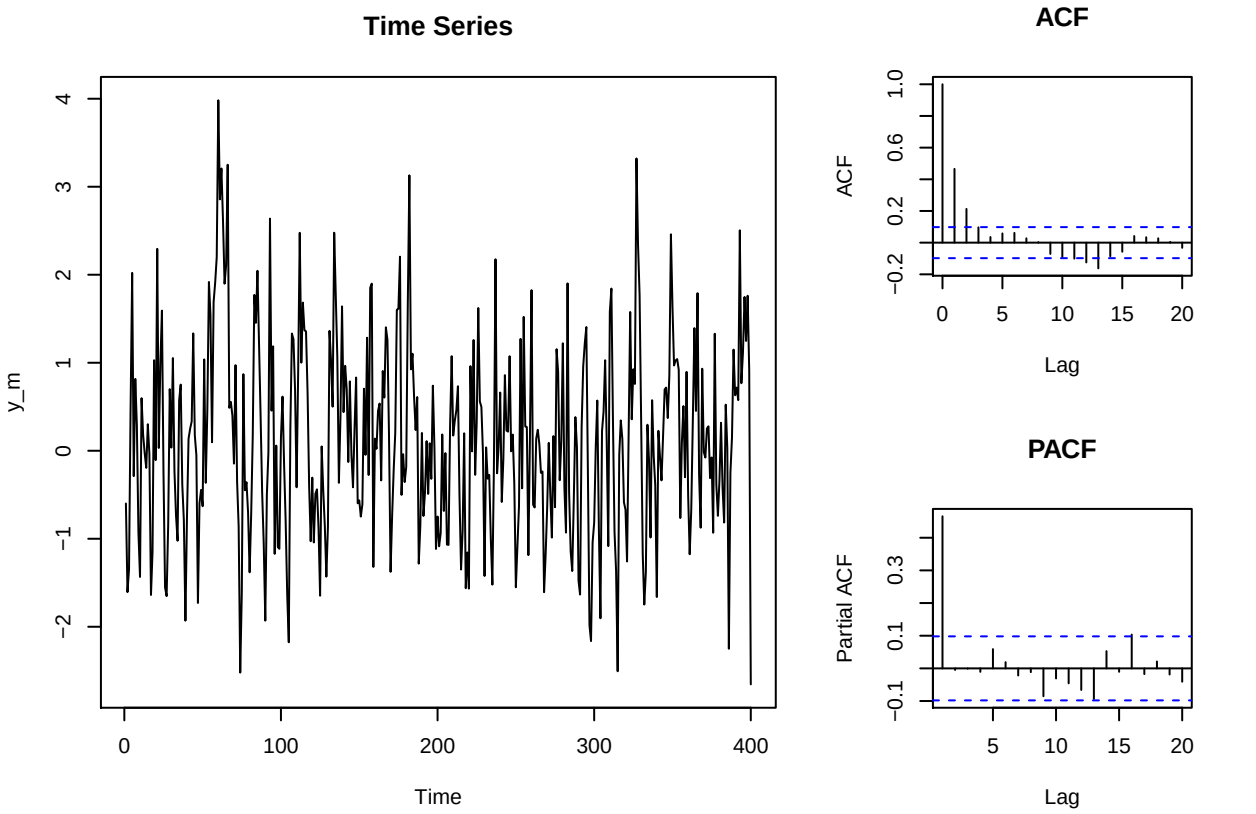
```
y_m <- y[1:400] # 1:400 = c(1,2, ... , 399, 400)
y_f <- y[401:404]
length(y) # could be written in a way to allow for different settings, do you see how?
```

```
## [1] 404
```

It's good practise to sort of say aloud what a line of code tells R to do, and when using the assignment operator (<-) you start on the right hand side of it. So the first line tells R to take the first 400 observations of y, then assign them to the object of y_m. The second line of code tells R to take observations 401 to 404 of y and store them in y_f. For practise you could try the same procedure but using the length() function and the fact that you want to forecast 4 observations.

The first step that I take now is to determine what type of process that generated my observed 400 observations. That is done in the same way as you did in the first assignment. Now recall how we use the layout function. In the matrix argument below it says that plot 1 should take positions row = 1,2 & column = 1,2. Then plot 2 takes position row = 1 & column = 3 and plot 3 takes position row = 2 & column = 3. When calling the layout function likes this, R won't forget how you told it to split up the plot window the next time you plot, so it's a good idea to end the code below with setting it back to default (1 plot takes the entire space).

```
# split up plot window
layout(matrix(c(1, 1, 2,
                1, 1, 3), nrow=2, byrow=TRUE))
ts.plot(y_m, main = "Time Series") # time series plot, plot 1 in matrix argument
acf(y_m, lag.max = 20, type = "correlation", plot = T, main = "ACF") # ACF, plot 2 in matrix argument
acf(y_m, lag.max = 20, type = "partial", plot = T, main = "PACF") # PCAF, plot 3 in matrix argument
```




```
par(mfrow = c(1,1)) # set plot window to default
```

By now you should be familiar with how to determine what type of process that generated the data based on the above plots. So if I determine this to be the fingerprint of an AR(1) the next step would be to estimate the model. This is done using the *arima*() function. If you type ?*arima* you find that you should supply the function with a univariate time series, and also the model specification. This is done through the order argument of the function. In the description you should find that the argument consists of (p, d, q) which are integers specyfing the AR order, the degree of differencing, and the MA order respectively. So I estimate my AR(1) as follows

```
m <- arima(y_m, order = c(1, 0 , 0)) # AR order = 1, degree of differencing = 0, MA order = 0
m
```

```
##
## Call:
## arima(x = y_m, order = c(1, 0, 0))
##
## Coefficients:
##          ar1  intercept
##       0.4726     0.1183
## s.e.  0.0444     0.0918
##
## sigma^2 estimated as 0.9413:  log likelihood = -555.61,  aic = 1117.23
```

Based on the above output I find that $\hat{\phi}_1 = 0.4726$ and that the standard error of the estimate is 0.0444. Before moving on I will save the residuals from the model estimation aswell as $\hat{\sigma}_u^2$

```r
u <- residuals(m)
(sigma2 <- m$sigma2)
```

```
## [1] 0.9413497
```

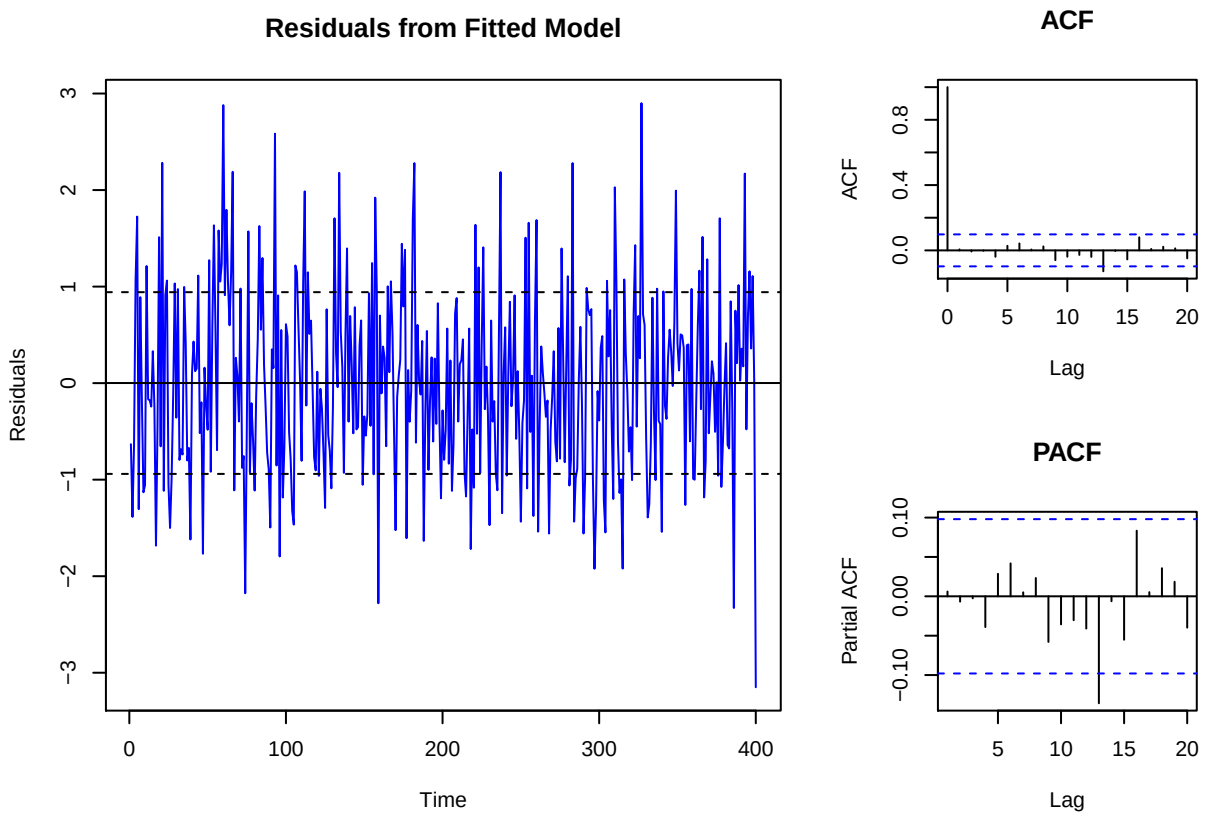
```r
#R hack: paranthesis around the whole expression prints the results
#directly without you having to type in sigma2 afterwards to check
```

Now we're going to have to inspect the residuals. The question we want to answer is whether we have choosen model correctly or not. This plot is pretty much the same as before, but I will add a solid line representing $\bar{\hat{u}}$ and dotted lines representing $\bar{\hat{u}} \pm \hat{\sigma}_u^2$

```r
# split up plot window as before
layout(matrix(c(1, 1, 2,
                1, 1, 3), nrow=2, byrow=TRUE))
ts.plot(u, ylab = "Residuals", col = "blue", main = "Residuals from Fitted Model")
abline(a = mean(u), b = 0) # adds horizontal line with mean(u) as intercept and 0 slope
abline(a = mean(u) + sigma2, b = 0, lty="dashed") # same as above + sigma2
abline(a = mean(u) - sigma2, b = 0, lty="dashed") # same as above - sigma2

acf(u, lag.max = 20, type = "correlation", plot = T, main = "ACF") # ACF

acf(u, lag.max = 20, type = "partial", plot = T, main = "PACF") # PCAF
```




```r
par(mfrow = c(1,1)) # set plot window to default
```

Using this information to assess model fit is up to you, keep in mind that you should ignore ACF at t = 0. In order to get access to the actual ACF values for testing, you make the same call as above but set plot =

FALSE, or plot = F. Ignore ACF at t = 0.

```
(u_acf <- acf(u, lag.max = 20, type = "correlation", plot = F) )
```

```
##
## Autocorrelations of series 'u', by lag
##
##      0      1      2      3      4      5      6      7      8      9
##  1.000  0.006 -0.007 -0.003 -0.039  0.028  0.043  0.005  0.024 -0.060
##     10     11     12     13     14     15     16     17     18     19
## -0.039 -0.028 -0.040 -0.128 -0.004 -0.055  0.080  0.009  0.022  0.012
##     20
## -0.049
```

The last part is to forecast the last 4 observations of the complete time series. This is done using the *predict*() function. We supply the function with the model object aswell as set *n.ahead* = 4, meaning we want to forecast 4 observations ahead

```
(preds <- predict(object = m, n.ahead = 4))
```

```
## $pred
## Time Series:
## Start = 401
## End = 404
## Frequency = 1
## [1] -1.19245324 -0.50124760 -0.17455165 -0.02013993
##
## $se
## Time Series:
## Start = 401
## End = 404
## Frequency = 1
## [1] 0.9702318 1.0731460 1.0948153 1.0995978
```

The object we created is of a very special type in R, called lists. These are usually a nightmare to work with in the beginning, but this one has named components so you can subset it the same way as you would a data frame

```
(y_pred <- preds$pred)
```

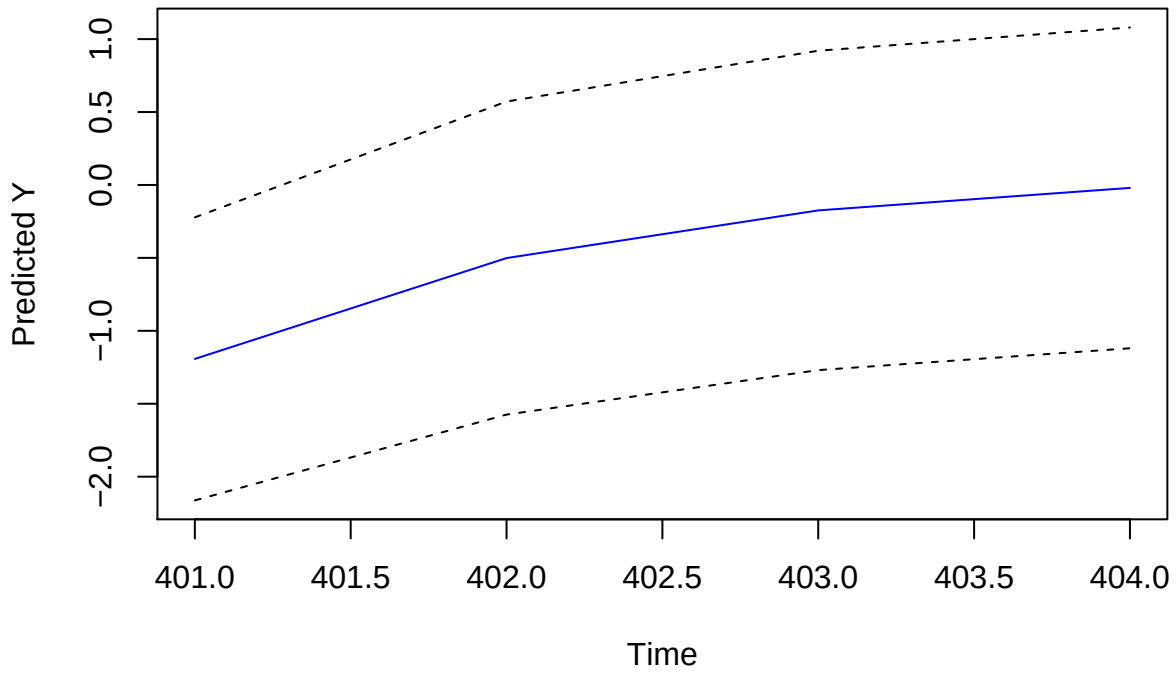
```
## Time Series:
## Start = 401
## End = 404
## Frequency = 1
## [1] -1.19245324 -0.50124760 -0.17455165 -0.02013993
```

```
(se_y_pred <- preds$se)
```

```
## Time Series:
## Start = 401
## End = 404
## Frequency = 1
## [1] 0.9702318 1.0731460 1.0948153 1.0995978
```

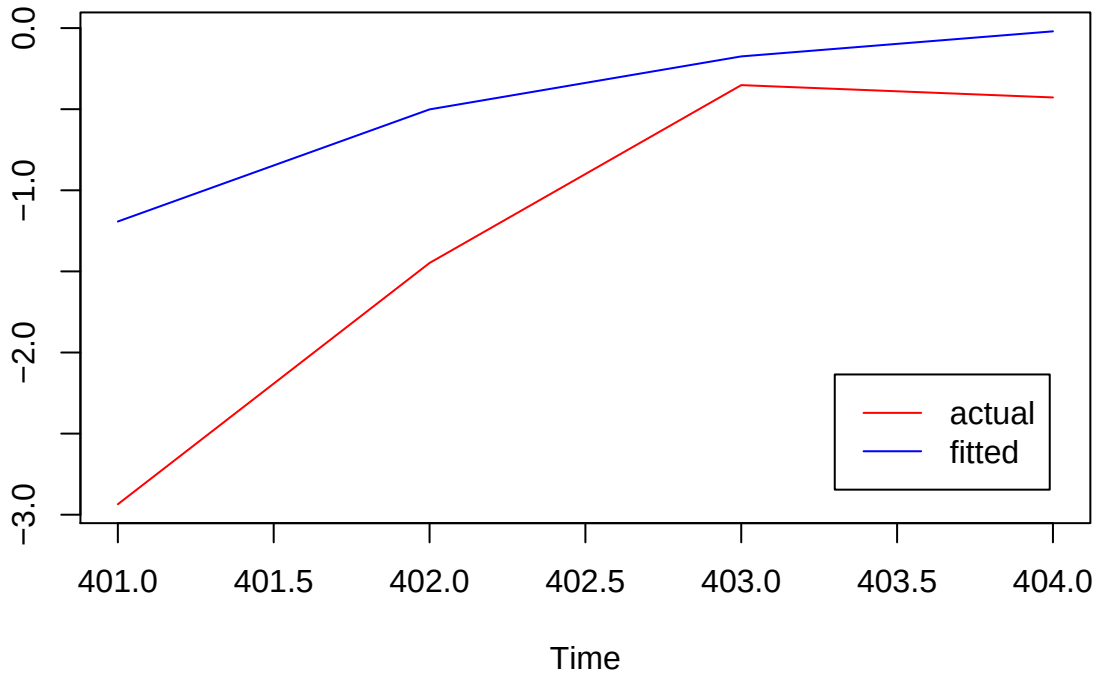
Now we will investigate the predicted y's by ploting them together with a confidence interval of the estimates. We have used *ts.plot*() quite extensively up until now. One feature of it that we have not yet explored however is the fact that we can add several time series to the same plot. Below I add the predicted y along with predicted y - se_y_pred and predicted y + se_y_pred

```r
ts.plot(y_pred, y_pred + se_y_pred, y_pred - se_y_pred, # time series
        ylab = "Predicted Y", # header on y-axis
        main = "", # plot header
        lty=c("solid", "dashed", "dashed"), col = c("blue", "black", "black"))
```



Next lets compare the actualy y's with predicted y's by plot. As always, adding a legend to a plot that hasn't been created using the package ggplot2 is a hassle. When making time series where you need to add a legend you can start by ploting and look for where you have space to put the legend. Here I would like to put it in the bottom right corner. This is close to max(time) and min(actual y), but not quite. After trial and error i find that good coordinates for the legend is $x = 404 - 0.7$ and $y = min(y\_f) + 0.8$

```r
# compare fitted and actual
ts.plot(y_f, y_pred, col = c("red", "blue"))
# add legend to the plot
legend(x = 404-.7, y = min(y_f) + .8,
       legend = c("actual", "fitted"),
       col = c("red", "blue"), lty=c(1,1))
```



Lastly we need some measurements for evaluationg model fit. This is easily done by hand and you are free to copy my functions written below. You could probably find some package that would do this, but It's very good practise for you to look at the wikipedia page for each of these fit indecies and compare with my functions.

```
# Root Mean Square Error (RMSE)
sqrt(mean((y_f-y_pred)^2))
```

```
## [1] 1.016361
```

```
# Mean Absolute Error (MAE)
mean(abs(y_f-y_pred))
```

```
## [1] 0.8186262
```

```
# Mean Absolute Percentage Error (MAPE)
mape <- 0
for(i in 1:4) {
  mape <- mape + abs((y_f[i] - y_pred[i])/y_f[i])
}
(100/4)*mape
```

```
## [1] 67.60713
```