TSA Assignment 9

Christian Hilscher - 1570550

18/5/2020

Question 15

DGP

For gathering the data the following function is used. The *models* input is a list containing strings to identify the models as given in the assignment. It first generates the errors drawn from different distributions and then uses the function *generate_x* to provide the finished time series. The results are then stored in a dictionary with the keys being the individual model names.

```
DGP <- function(tslength, samplesize, b, models){

  storage <- dict()
  for (model in models){
    storage[[model]] <- vector()

    # Drawing errors
    for (i in seq(1, samplesize)){
      if (model=="i"){
        errors <- rnorm(1001)
      }else if (model=="ii"){
        errors <- rt(1001, 5)
      }else if(model=="iii"){
        errors <- runif(1001)
      }else{
        stop('Please provide one of the models: i, ii, iii')
      }

      # Getting the time series with the correpsonding errors
      storage[[model]] <- rbind(storage[[model]],
                                generate_x(tslength, errors, b))
    }
  }
  return(storage)
}

generate_x <- function(tslength, errors, b){

  e_shifted <- shift(errors,1)
  x <- (errors+b*e_shifted)[2:1001]
  return(x)
}
```

For setting up the whole process I run the following specifications. Here I also provide the true variances of the individual models because I need them later on. The vector conatining the λ_j made according to the slides, nothing fancy there.

```r
TS_length <- 1000
Sample_size <- 1000
b <- 0.25
models <- cbind('i', 'ii', 'iii')
sigma_list <- cbind(1, 5/3, 1/12)

data <- DGP(TS_length, Sample_size, b, models)
M <- floor((TS_length-1)/2)
lambdas <- (2*pi/TS_length) * seq(1:M)

# Making space for end results
df_final <- data.frame()
```

Frequency Domain

Theoretical Setup

From the $MA(1)$ model $x_t = \epsilon_t + b\epsilon_{t-1}$ I can infer the following properties:

$$\begin{aligned}
I_x(\lambda_j) =& \frac{1}{2\pi T}\left(\left(\sum_{t=1}^{T} x_t cos[\lambda_j(t-2)]\right)^2 + \left(\sum_{t=1}^{T} x_t sin[\lambda_j(t-2)]\right)^2\right) \\
f_x(\lambda) =& (1+b^2+2bcos(\lambda))\frac{\sigma^2}{2\pi} \\
I_M(b,\sigma^2) =& \sum_{j=1}^{M}\left[-ln\left((1+b^2+2bcos(\lambda))\frac{\sigma^2}{2\pi}\right) - \frac{I_x}{(1+b^2+2bcos(\lambda))\frac{\sigma^2}{2\pi}}\right] \qquad (*)
\end{aligned}$$

The aim is to maximise $(*)$ for which I first take the derivative and single out σ^2 and plug it back into $(*)$ such that I only have to optimize over b.

Taking the derivative, setting it equal to zero and solving for σ^2 yields:

$$\begin{aligned}
\frac{\partial I_M}{\partial \sigma^2} &= \sum_{j=1}^{M} -\frac{1}{\Delta\frac{\sigma^2}{2\pi}}\Delta 2\pi + \frac{1}{2\pi}\frac{I_x}{\Delta}\frac{1}{\sigma^4} \overset{!}{=} 0 \\
\Leftrightarrow \quad & \sum_{j=1}^{M}\frac{4\pi}{\sigma^2} = \sum_{j=1}^{M}\frac{1}{2\pi}\frac{I_x}{\Delta}\frac{1}{\sigma^4} \\
\Rightarrow \quad & \sigma^2 = 2\pi\frac{1}{M}\sum_{j=1}^{M} I_x\frac{1}{\Delta}
\end{aligned}$$

with $\Delta = (1+b^2+2bcos(\lambda))$ and I_x as defined above. One can insert this into $(*)$ which is now only a function of b and will be maximised over b only. Inserting σ^2 leads to cancellation of the 2π in both cases so from now on $\sigma^2 = \frac{1}{M}\sum_{j=1}^{M} I_x\frac{1}{\Delta}$. Pulling apart the ln then yields:

$$I_M(b) = \sum_{j=1}^{M}\left[-ln(\Delta) - ln(\sigma^2) - \frac{I_x}{\Delta}\frac{1}{\sigma^2}\right] \qquad (+)$$

Coding Part

Here I define the functions that eventually make up the objective function (+) which I maximise. I tried to vectorize every function as far as possible to make the processing time somewhat shorter.

Getting the Δ as describes above is achieved by running

```r
delta <- function(b, lambdaj){
  val <- (1 + b**2 + 2*b*cos(lambdaj))
  return(val)
}
```

This function can also take in the whole vector containing all λ_j and then returns an array of equal size with the corresponding Δs

The individual I_x are a function of λ_j and the provided time series x. This function takes the vector x and generated two vectors with the *sin* and *cos* respectively. Then I multiply elementwise and at the end take the sum over the whole vector.

```r
Ix <- function(lambdaj, x){

  bigT <- length(x)
  little_t_seq <- seq(bigT)-1

  prefactor <- 1/(2*pi*bigT)


  sequence_sin <- cos(lambdaj * little_t_seq)
  sequence_cos <- cos(lambdaj * little_t_seq)

  val <- prefactor*(sum(x*sequence_cos)**2 + sum(x*sequence_sin)**2)
  return(val)
}
```

Coming to σ^2, the corresponding function takes as inputs the list of λ_j, a specific b and x. Then again $sumoverIx$ and $sumoverdelta$ are vectors of length M which I sum at the end.

```r
sigma <- function(lambda_list, b, x){

  bigM <- length(lambda_list)

  sumoverIx <- sapply(lambda_list, Ix, x=x)
  sumoverdelta <- delta(b=b, lambda_list)

  val <- (1/bigM) * sum((sumoverIx/sumoverdelta))
  return(val)
}
```

The objective function (+) is finally given by

```r
objf <- function(b, lambda_list, x){

  firstterm <- delta(b, lambda_list)
  secondterm <- sigma(lambda_list, b, x)

  thirdterm_Ix <- sapply(lambda_list, Ix, x)

  tmp <- -log(firstterm) - log(secondterm) - (thirdterm_Ix/firstterm)*(1/secondterm)
  val <- sum(tmp)

  return(val)
}
```

Maximising I_M

The maximisation is then done by first defining a grid with $\hat{b} \in (-1, 1)$ since if $|\hat{b}| > 1$ the process would not be stable. For this I run the following loop where for first the data is taken from the dictionaries I computed before.

I find the optimal b b_{opt} by maximising the objective function over the grid defined above. For calculating the corresponding σ^2 I plug the b_{opt} into the *sigma* function and then multiply it by 2π to get the correct value. The MSE of the estimated parameters b_{opt} and σ_{opt} is given by taking their respective biases to the power of two and adding the variance.

```
for (model in models){
  pos <- which(models==model)

  data_used <- data[[model]]
  sample_size <- dim(data_used)[1]
  ts_length <- dim(data_used)[2]

  df <- data.frame()

  for (s in seq(sample_size)){

    res <- sapply(b_list, objf, lambda_list=lambdas, x=data_used[s,])

    b_opt <- b_list[which.max(res)]
    sigma_opt <- sigma(lambdas, b_opt, x=data_used[s,]) * 2 * pi

    df <- rbind(df, cbind(b_opt, sigma_opt))
  }
  colnames(df) <- c("b", "sigma")

  bias_b <- (mean(df$b) - b)*ts_length
  bias_sigma <- (mean(df$sigma) - sigma_list[pos])*ts_length

  # Getting the values for MSE
  sigma_b <- var(df$b)
  sigma_sigma <- var(df$sigma)

  df_final <- rbind(df_final, cbind(bias_b, bias_sigma, sigma_b, sigma_sigma))
}
```

Time Domain

Theoretical Setup

First getting the conditional mean and variance of x_{t+1}:

$$\begin{aligned}
E[x_{t+1}|I_t] =& E[e_{t+1} + be_t|I_t] \\
=& E[e_{t+1}|I_t] + E[be_t|I_t] \\
=& be_t \\
&\text{and} \\
Var(x_{t+1}|I_t) =& E[x_{t+1}^2|I_t] - E[x_{t+1}|I_t]^2 \\
=& E[e_{t+1}^2 + 2be_{t+1}e_t + b^2e_t^2|I_t] - b^2e_t^2 \\
=& e_{t+1}^2 = \sigma^2
\end{aligned}$$

The conditional Loglikelihood is then given by

$$\begin{aligned}
LLH^c(b, \sigma^2) =& \sum_{t=1}^{T} ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{1}{2\sigma^2}\left(x_t - E[x_t|I_t]\right)^2 \\
=& (T-1)ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{1}{2\sigma^2}\sum_{t=1}^{T} e_t^2
\end{aligned}$$

Since I only observe x_t I assume that $e_0 = 0$ and approximate e_t by $\hat{e}_t$

$$\begin{aligned}
\hat{e}_0 =& 0 \\
\hat{e}_1 =& x_1 \\
\hat{e}_2 =& x_2 - bx_1 \\
\hat{e}_3 =& x_3 - bx_2 \\
&\vdots \\
\hat{e}_t =& \sum_{s=0}^{t} x_{t-s}b^s(-1)^s
\end{aligned} \tag{-}$$

As in the frequency domain, I concentrate σ^2 out such that the objective function to be maximised is only depending on one parameter, b.

$$\begin{aligned}
\frac{\partial LLH^c}{\partial\hat{\sigma}^2} =& \sum_{t=1}^{T} \sqrt{2\pi\hat{\sigma}^2}\left(-\frac{1}{2\sqrt{2\pi}}\frac{1}{\hat{\sigma}^2\sqrt{\hat{\sigma}^2}}\right) + \left(\frac{1}{2\hat{\sigma}^4}\hat{e_t}^2\right) \overset{!}{=} 0 \\
\Leftrightarrow \quad & \sum_{t=1}^{T}\frac{1}{2\hat{\sigma}^2} = \sum_{t=1}^{T}\frac{1}{2\hat{\sigma}^4}\hat{e_t}^2 \\
\Leftrightarrow \quad & \hat{\sigma}^2 = \frac{1}{T-1}\sum_{t=1}^{T}\hat{e_t}^2
\end{aligned}$$

For all three models (i, ii, iii) I assume the PDF of a normal distribution to see how the estimator compares to the one from the spectral density when having different distributions of the erros.

Coding

For this part I switched to Python and vectorized everything which in the end was faster by a factor of 10 compared to R.

As a start I calculate $(-)$. I represent the x_{t-s} by a lower triangular matrix where the first column only contians x_0 and then only zero. Column 2 then has $\begin{pmatrix} x_1 \\ x_0 \end{pmatrix}$ followed by zeros and so on.

```
def make_xmatrix(x):

  n = len(x)
  xmat = np.zeros((n,n))

  for i in range(n):

      xtmp = x[0:n-i]

      tmp = np.zeros(n)
      tmp[n-len(xtmp):] = xtmp

      xmat[i,:] = tmp
  return xmat
```

The corresponding vector of b with the alternating signs is coded by

```
def make_bvector(b, x):
    n = len(x)
    bvector = np.repeat(b, n) ** np.arange(n)
    bvector = bvector * ((-1) ** np.arange(n))

    return bvector
```

I multiply that matrix and the vector elementwise to get a matrix of $\hat{e}$. Taking the sum over the individual columns then results in the vector of $\hat{e}$:

```
def make_ehat_mat(b, x):
    n=len(x)

    bvec = make_bvector(b, x)
    xmat = make_xmatrix(x)

    ehat_mat = np.empty((n,n))
    for i  in np.arange(n):
        ehat_mat[:,i] = np.multiply(xmat[:,i], bvec)
    return ehat_mat

def make_ehat(ehatmat):

    n = ehatmat.shape[1]
    res = np.empty(n)
    for i in range(n):
        res[i] = np.sum(ehatmat[:,i])
    return res
```

With this vector of $\hat{e}_t$ the corresponding $\hat{\sigma}^2$ are calculated with

```
  def get_sigmahat(b,x):
    n = len(x)

    ehatmat = make_ehat_mat(b,x)
    eh = make_ehat(ehatmat)

    sigma = (1/(n-1)) * np.sum((eh**2))
    return sigma
```

The objective function is given by $objf\_MLE$ and the optimisation is done by the function called *calc*.

```
  def objf_MLE(b, x):
    n = len(x)
    sigmahat = get_sigmahat(b,x)
    logpart = (1/np.sqrt(2*np.pi*sigmahat))

    ehatmat = make_ehat_mat(b,x)
    eh = make_ehat(ehatmat)

    value = (n-1) * np.log(logpart) - (1/(2*sigmahat)) * np.sum((eh**2))
    return value

def calc(blist, x):
    samplesize = np.shape(x)[0]
    res_df = np.empty((samplesize, 2))

    for j in np.arange(samplesize):
        data_used = x[j,:]

        res_tmp = np.empty(len(blist))

        for i in np.arange(len(blist)):

            b_tmp = blist[i]
            res_tmp[i] = objf_MLE(b_tmp, data_used)

        b_opt = blist[np.where(np.max(res_tmp)==res_tmp)][0]
        sigma_opt = get_sigmahat(b_opt, data_used)

        res_df[j,:] = [b_opt, sigma_opt]
    return res_df
```

Results

The values are then:

Table 1: Values from the frequency domain

	bias of $\hat{b}$	bias of $\hat{\sigma}^2$	MSE $\hat{b}$	MSE $\hat{\sigma}^2$
model *i*	4.40	-1.97	2.06	3.96
model *ii*	0.60	1.71	2.14	28.55
model *iii*	-1.55	-0.22	2.11	0.02

Table 2: Values from the time domain

	bias of $\hat{b}$	bias of $\hat{\sigma}^2$	MSE $\hat{b}$	MSE $\hat{\sigma}^2$
model i	3.10	-0.65	1.22	2.05
model ii	-0.05	3.1	1.17	22.21
model iii	404	167	163	27.95

In model i where the errors are drawn from a standard normal distribution the MLE estimator performs better than the frequency domain estimation. The same is still roughly true for the Student's t-distribution which is pretty similar to the standard normal.

As soon as the errors are drawn from a distribution substantially different as assumed in the MLE, this estimator performs very poorly in contrast to the frequency domain estimation. This estimator's performance does not depend on the distribution of the errors.

Suming up, when the distribution of the errors is known, the MLE estimator does a better job than using the Whittle estimator which still performs fairly good. With the distribution of the errors deviating from the assumed one, the MLE estimation should not be the preferred one. As most things in life, there is a trade-off striking again. Going for the more exact MLE estimation is coming with the drawbacks of having a really bad estimator when choosing the wrong underlying distribution. The Whittle estimator on the other hand is independent of the distribution but also less accurate compared to the optimal MLE.

Question 16

DGP

Prelinimaries

For generating the data, I resort to using the following formula

$$\begin{aligned} y_t &= \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} && (1) \\ \text{with } \psi_j &= \frac{\Gamma(j+d)}{\Gamma(j+1)\Gamma(d)} \\ \text{and } \Gamma(z) &= \int_0^\infty \underbrace{x^{z-1}e^{-x}}_{f(x,z)} dx, \quad \text{where } Re(z) > 0 \end{aligned}$$

However I approximate ψ_j by

$$\psi_j \sim \frac{j^{d-1}}{\Gamma(d)}$$

Since I cannot really take the sum in (1) to ∞, I take it up to M+T and will later discard the first M observations. here I code the function f and then the integral over that function. Taking $d = 0$ makes it impossible to start at the lower bound of zero since then I'd be dividing by zero. This can be seen when taking the antiderivative of f. That's why I start with the lwoest possible value which is 1e-3 in that case.

```
f <- function(x, z){
  value <- x**(z-1)*exp(-x)
  return(value)
}

intf <- function(z){
  if (z != 0){
    value <- integrate(f, lower=0,upper=Inf, z=z)$value
  }else{
    value <- integrate(f, lower=1e-3,upper=Inf, z=z)$value
  }

  return(value)
}
```

For getting the ψ_j one needs to be carefull with the order of the signs and the expression changes depending on d being negative or positive.

```
psi <- function(j, d){

  if (d<0){
    upper <- j**(-d-1)
    lower <- intf(abs(d))
  }else{
    upper <- j**(d-1)
    lower <- intf(d)
  }

  value <- upper/lower
  return(value)
}
```

Getting the data

The final data gathering then happens according the following function

```
get_data <- function(overall_length, d_list){
  y <- matrix(nrow=overall_length, ncol=length(d_list))

  for (i in seq_along(d_list)){
    # Initiating space
    res <- rep(NA, length(overall_length))

    # Summing
    for (t in seq_len(overall_length)){
      val <- 0
      for (j in seq_len(t-1)){
        tmp <- psi(j,d_list[i]) * et[t-j]
        val <- val + tmp
      }
      res[t] <- sum(val)
    }

    y[,i] <- res
  }
  return(y)
}
```

Autocorrelations

For the autocorrelations I follow the slides and use

$$\begin{aligned}\gamma(0) =&\sigma^2 \frac{\Gamma(1-2d)}{\Gamma^2(1-d)} \\ \gamma(h) =&\frac{h-1+d}{h-d}\gamma(h-1)\end{aligned}$$

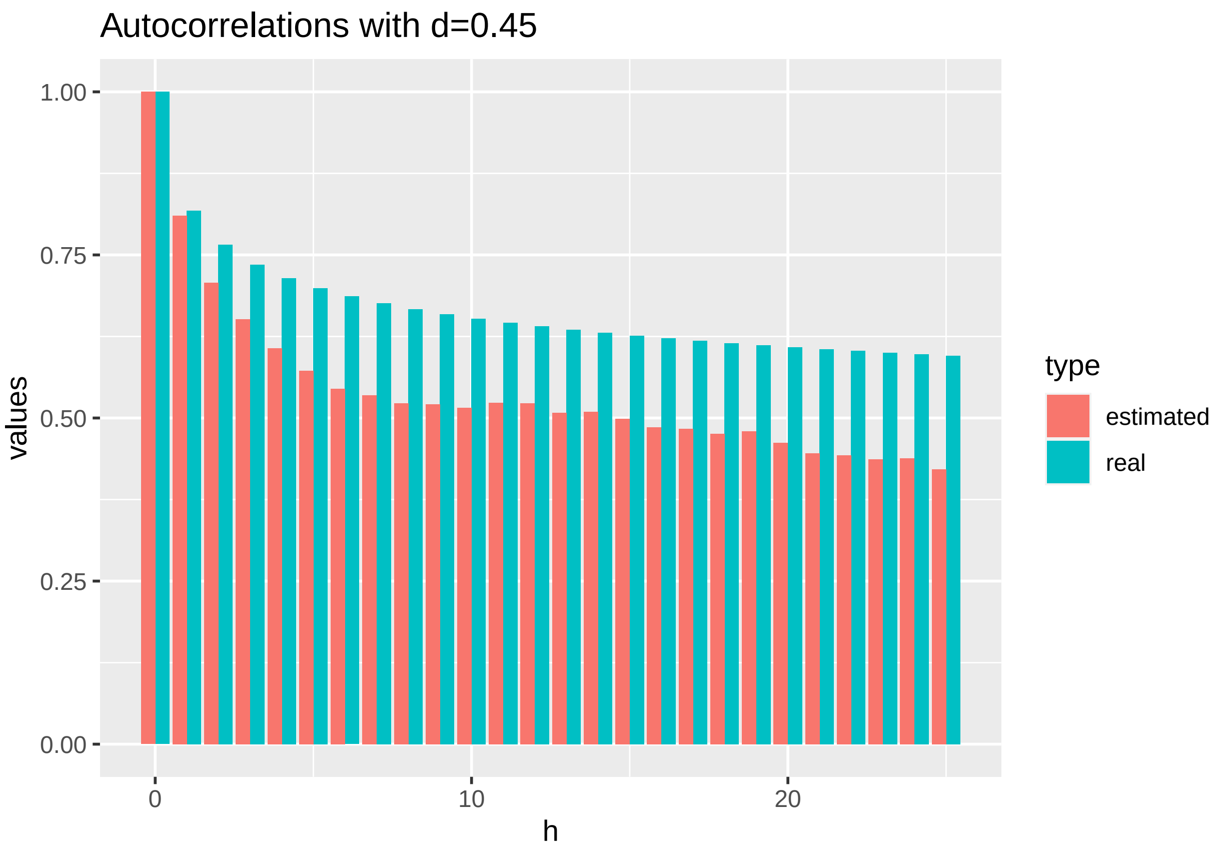
Getting $\gamma(0)$ is done by running

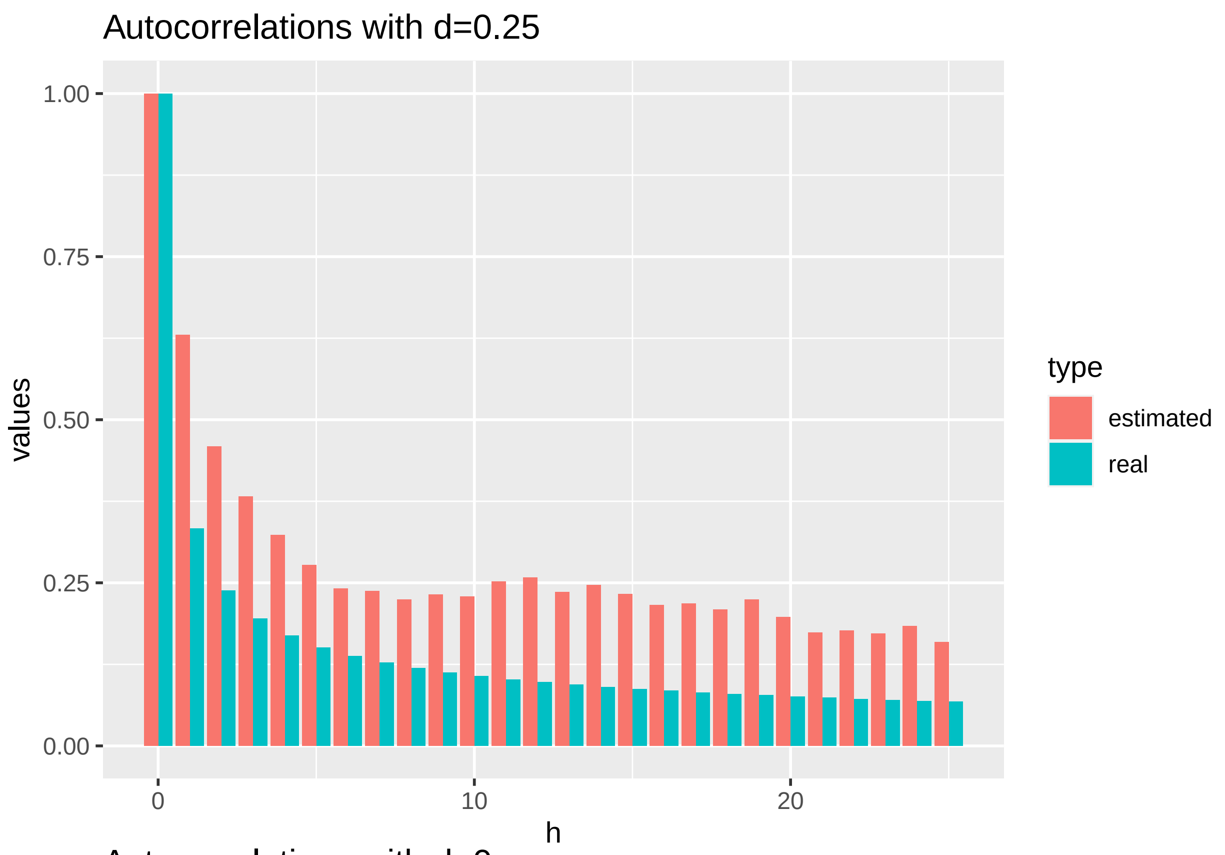
```r
get_gamma0 <- function(data, d_list, d){

  pos <- which(d_list==d)
  sigma2 <- var(data[,pos])
  upper <- intf(1-2*d)
  lower <- intf(1-d)**2

  final <- sigma2*(upper/lower)
  return(final)
}
```

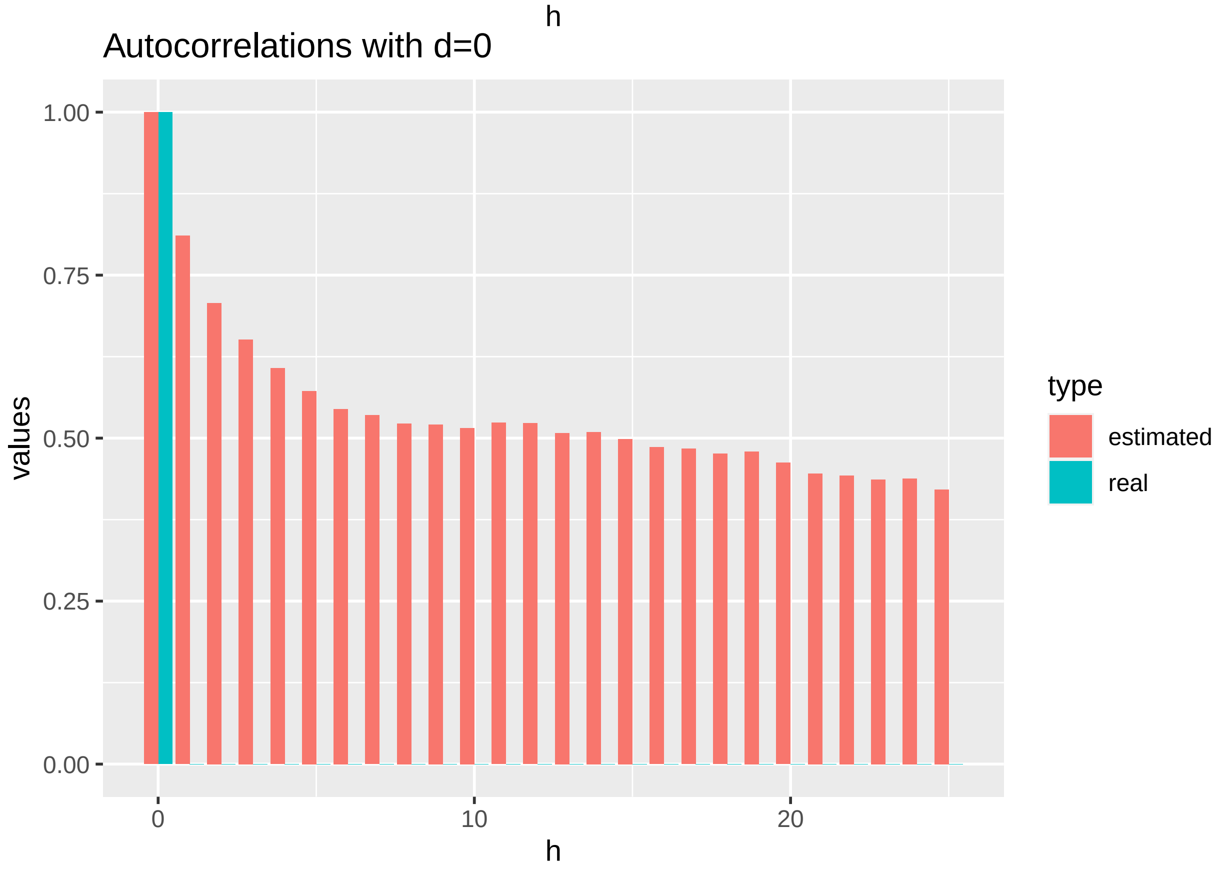
which is a function of all of the d considered, the actual d as well as the data. Somewhat clunky implementation but works for now. The variance is just teh estimated variance of the time series. \ The rest of the covariances and then subsequently the autocorrelations are given by this rather messy function

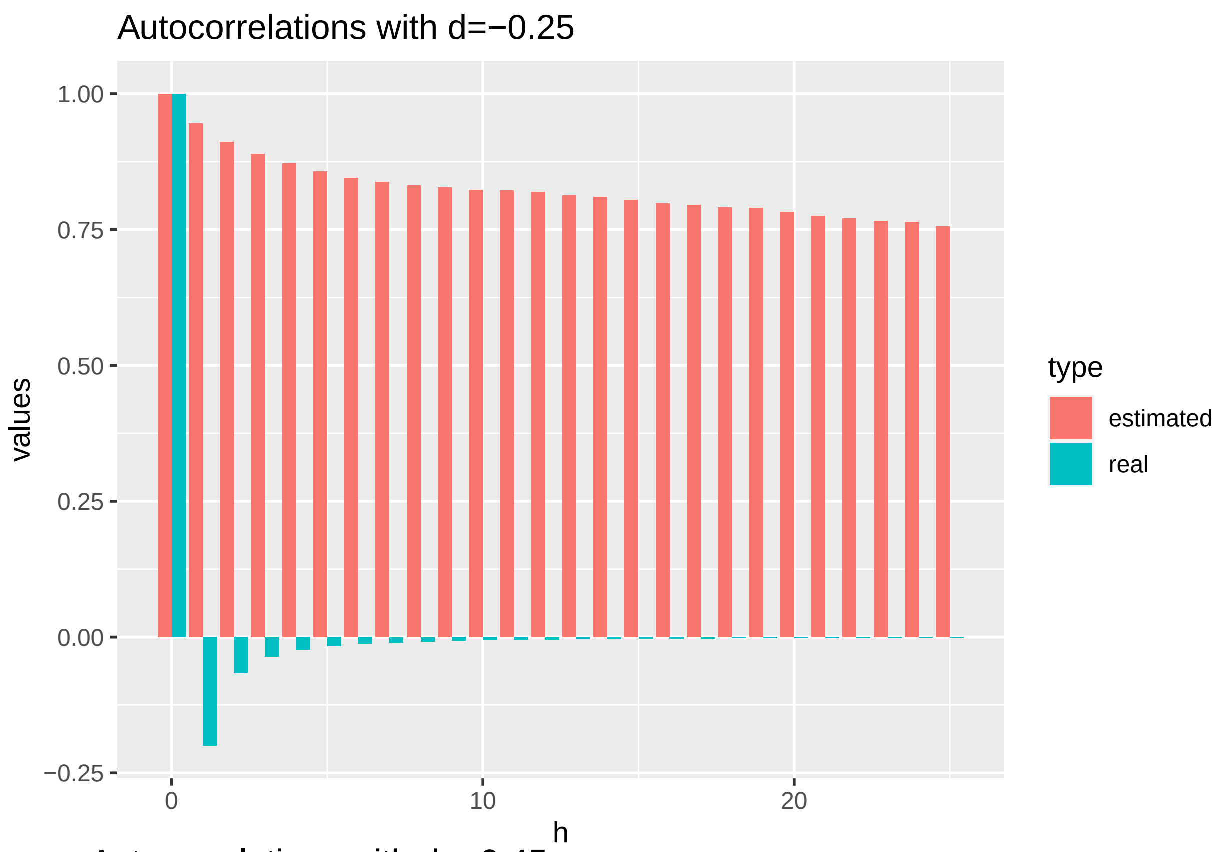
```r
get_corrs <- function(data, laglength, d_list){

  sto = dict()
  for (i in seq_along(d_list)){
    # Initiating storeage space
    lagframe <- data.frame()

    # Getting gamma(0) and building up
    gamma0 <- get_gamma0(data, d_list, d_list[i])
    lagframe <- cbind(0, gamma0)


    for (h in seq(laglength)){
      pre <- (h-1+d_list[i])/(h-d_list[i])
      gammah <- pre * lagframe[h,2]

      lagframe <- rbind(lagframe, cbind(h, gammah))
    }

    # Calculating the autocorrelation from the covariances
    lagframe[,2] <- lagframe[,2]/lagframe[1,2]
    colnames(lagframe) <- c("h", 'real')

    # Getting the estimated autocrrelations
    estimated <- acf(data[,i], lag.max=laglength,
                     plot=FALSE, demean = FALSE)$acf
    final <- cbind(lagframe, estimated)
    sto[[toString(i)]] <- data.frame(final)
  }

  return(sto)
}
```

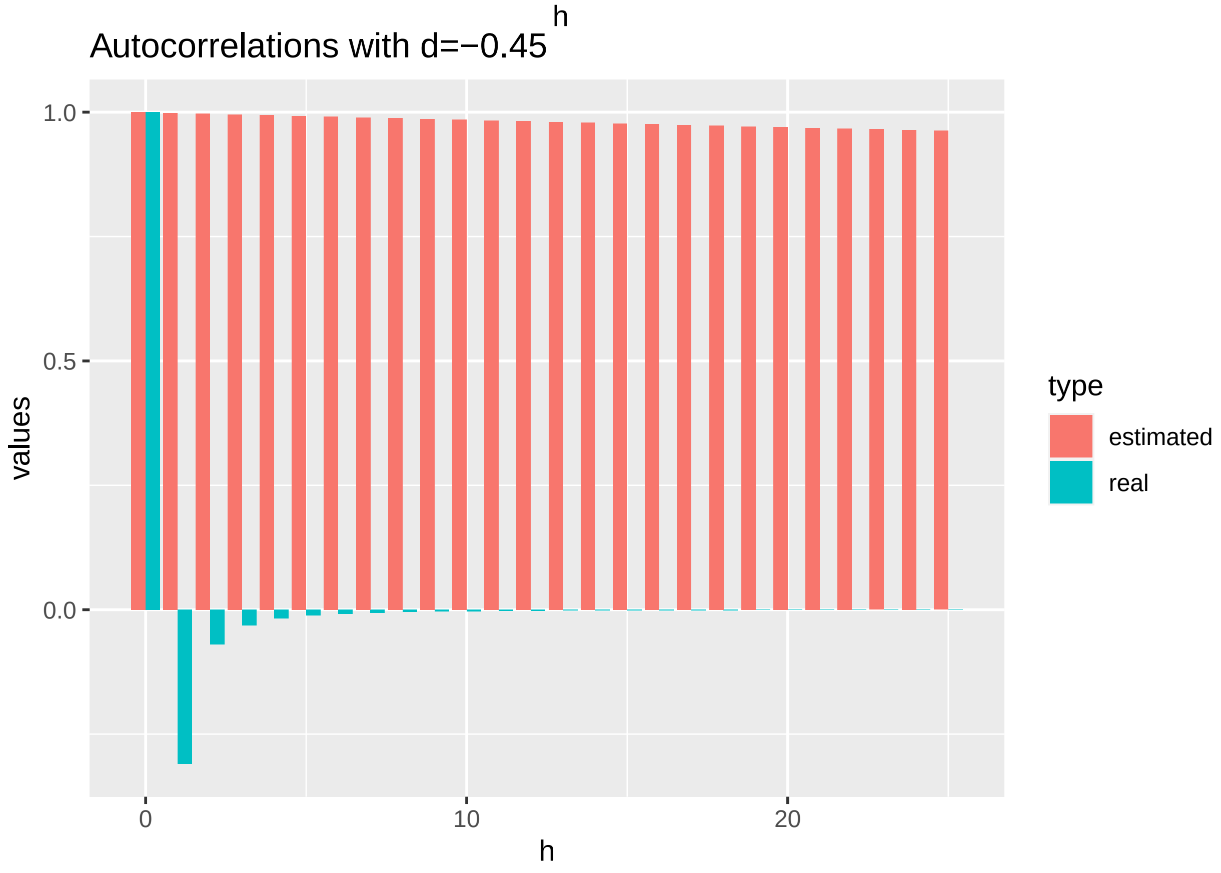
It also returns the estimated autocorrelations up to lag 25 which for which I used the standard R function *acf*. This is then saved in a dictionary. The results depending on the different d are given in the figures below. The values for *real* are calculated with the functions described above while the values for *estiamted* come from the *acf* function which estimates the autocorrelations of a given time series. The very first value is 1 all the time since it is the zero-th lag which simply $\frac{\gamma(0)}{\gamma(0)} = 1$.



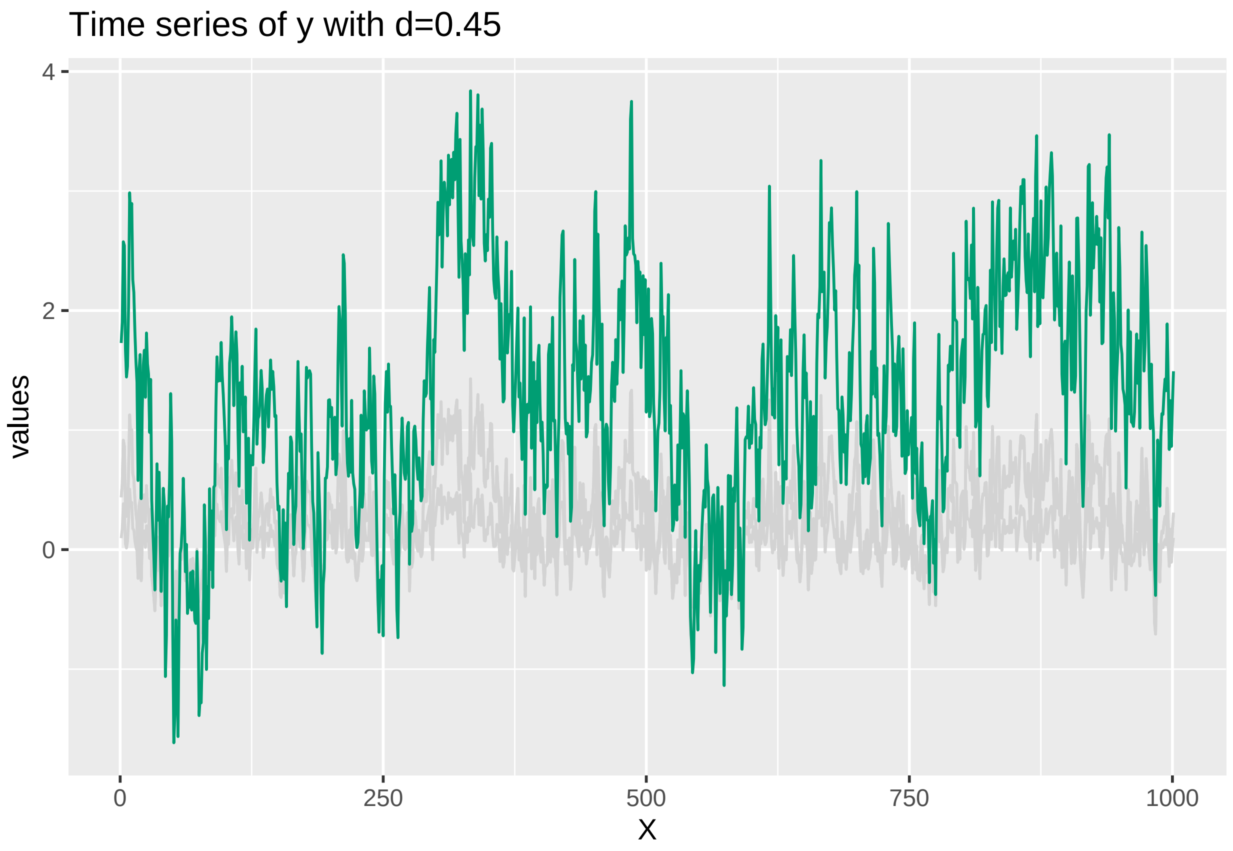

Autocorrelations with d=0.25
values
1.00
0.75
0.50
0.25
0.00
0
10
20
h
type
estimated
real


Autocorrelations with d=0
values
1.00
0.75
0.50
0.25
0.00
0
10
20
h
type
estimated
real

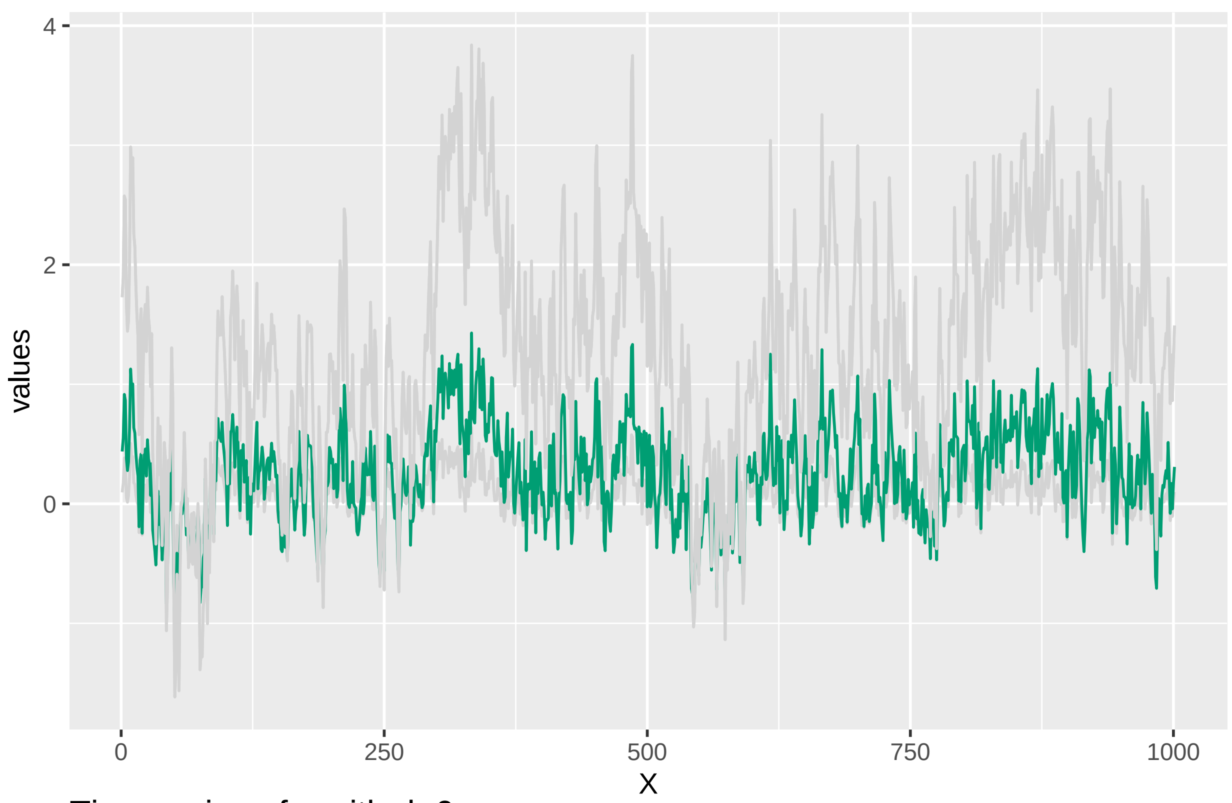




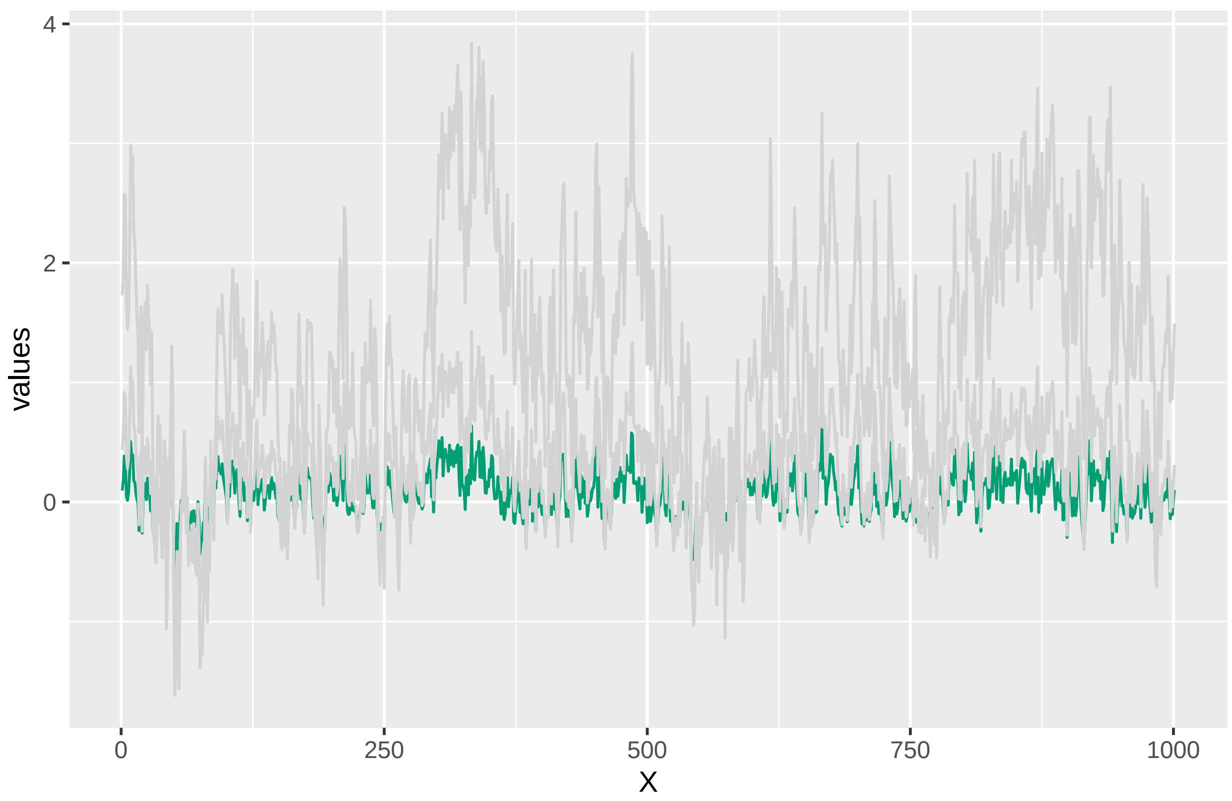
Now the plots with y_t. The interesting thing here is that the time series for $d = -0.45$ and $d = 0.45$ is basically the same, in the second case you almost cannot see it since it's overlayed by the other one. Apparently it does not make a difference to the time series whether it is fractually integrated by amount δ or $-\delta$. This can also be seen when deriving and coding the DGP. I always switch the sign such that the expression evaluated is the same.



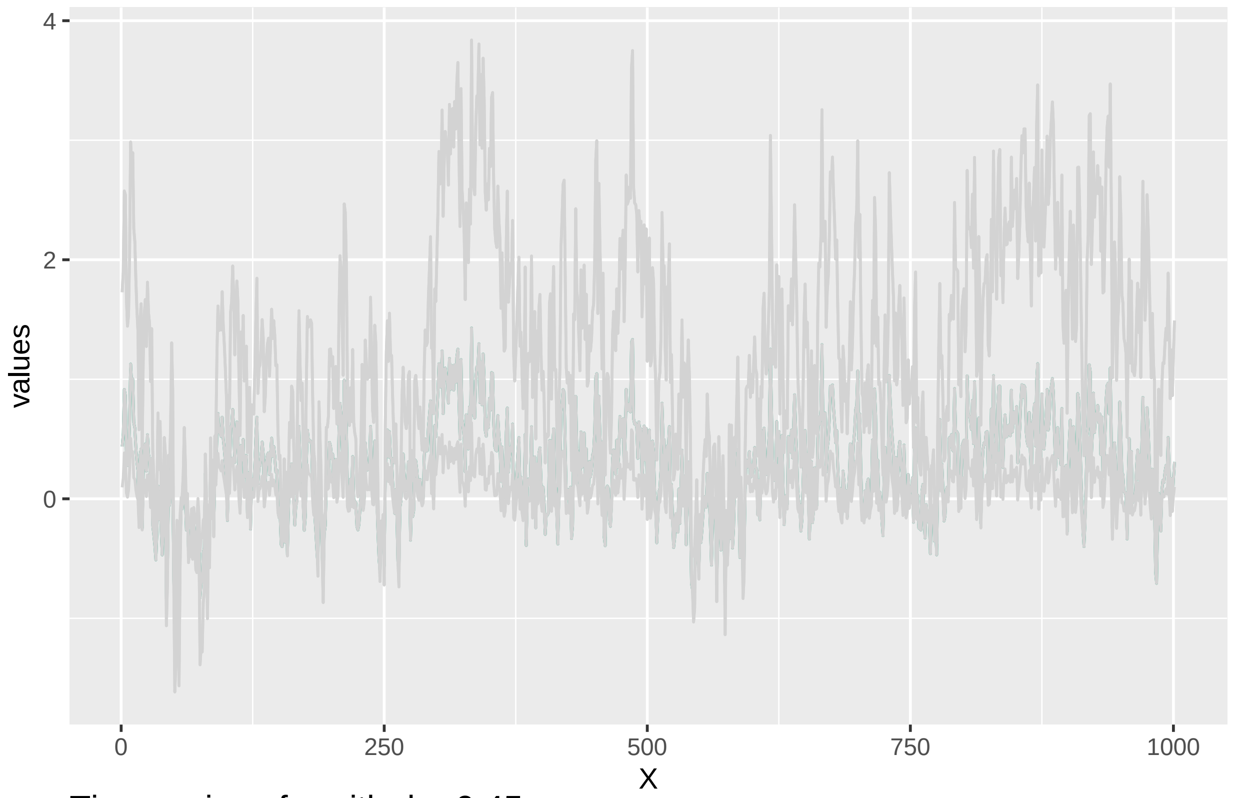
Time series of y with d=0.25



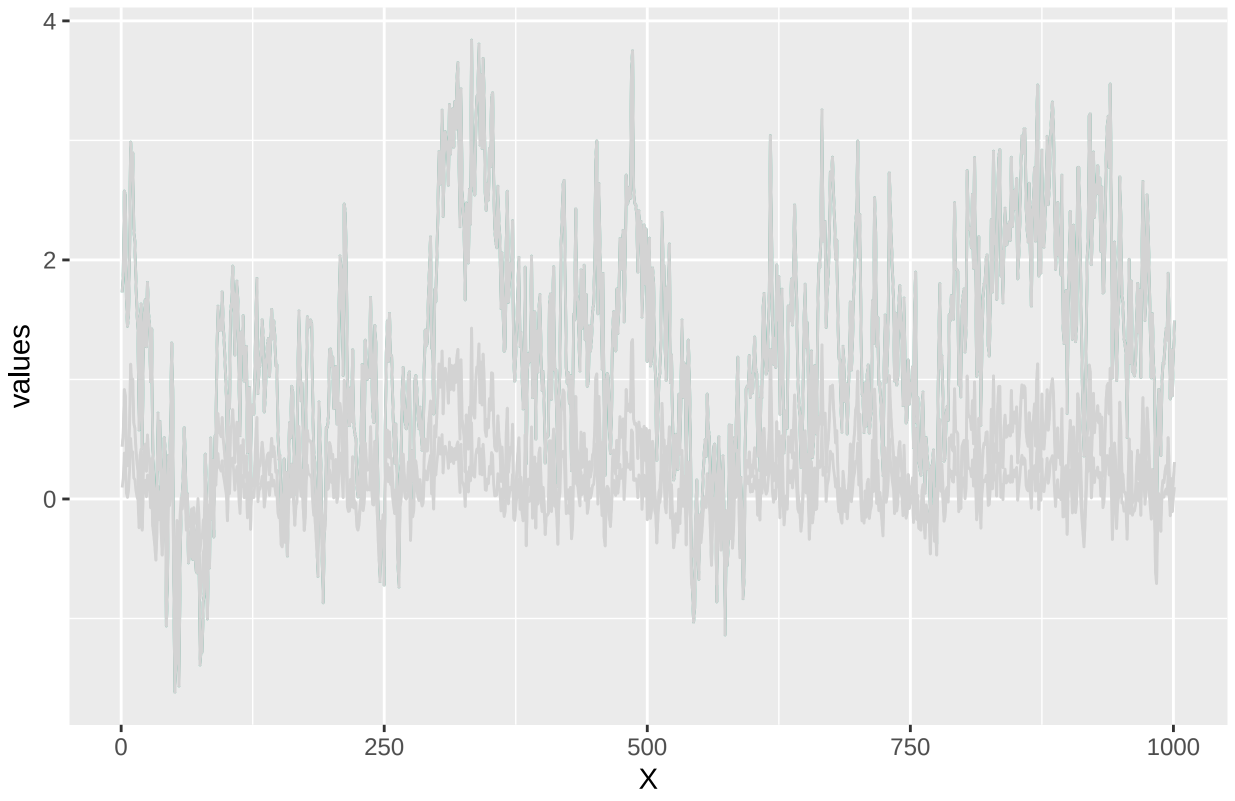
Time series of y with d=0



Time series of y with d=−0.25



Time series of y with d=−0.45



Appendix

All other functions used for both questions:

```
prepplot <- function(data){
  df_plot <- pivot_longer(data, -h, names_to = 'type', values_to = 'values')
  return(df_plot)
}

shift <- function(x, lag) {
  n <- length(x)
  xnew <- rep(NA, n)
  if (lag < 0) {
    xnew[1:(n-abs(lag))] <- x[(abs(lag)+1):n]
  } else if (lag > 0) {
    xnew[(lag+1):n] <- x[1:(n-lag)]
  } else {
    xnew <- x
  }
  return(xnew)
}
```

```
  def get_biases(dataf, model):
      bias_b = (np.mean(dataf[:,0]) - b)*1000
      bias_sigma = (np.mean(dataf[:,1]) - sigma_list[model-1])*1000

      sigma_b = np.var(dataf[:,0])
      sigma_sigma = np.var(dataf[:,1])

      MSE_b = ((bias_b/1000)**2 + sigma_b) * 1000
      MSE_sigma = ((bias_sigma/1000)**2 + sigma_sigma) * 1000

      return [bias_b , bias_sigma, MSE_b, MSE_sigma]
```