Topics in Time Series Analysis: Assignment 9

Lucas Cruz Fernandez

1544674, lcruzfer@mail.uni-mannheim.de

13^{th} of May 2020

Q15

To implement the Whittle estimator as discussed in class it is helpful to first single out the variance term σ^2 and then optimize the objective function as a function of parameter b only. We will first derive a function $sigma^2(b)$ which we can then plug back into our objective function $I_m(b, \sigma^2)$. The maximization (or minimzation of $-I_m$, which does not matter for the outcome) is then solved computationally on a grid of values for $b \in (-1, 1)$. We denote by $\Delta = (1 + b^2 + 2bcos(\lambda_j)$.

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

Note that we use the definition from slide 145 for $I_x(\lambda_j)$. Plugging this term for σ^2, which depends on b through Δ, back into the objective function yields our new objective function:

$$I_m(b) = \sum_{j=1}^{M} \left(-ln(\Delta) - ln\left(\frac{\sigma^2}{2\pi}\right) - \frac{I_x(\lambda_j)}{\Delta \frac{\sigma^2}{2\pi}} \right)$$

Solving this term and defining further $\Omega = \frac{\sigma^2}{2\pi}$ can now find the maximum of this function on a grid of b since Ω only depends on b as well.
We first define all necessary functions for solving this process. We begin Δ, I_x and $Omega$ as defined above.

```
def transfer(b, lambda_j):
```

```
    '''
    Transfer function of MA(1) process as given in class.
    '''
    return((1 + b**2 + 2*b*np.cos(lambda_j)))

def Ix(x_t, lambda_j):
    '''
    I_x as defined on slide 144.
    '''
    T = len(x_t)
    factor = 1/(2*np.pi*T)
    cos_term = np.sum([x_t[t] * np.cos(lambda_j*t) for t in range(T)])
    sin_term = np.sum([x_t[t] * np.sin(lambda_j*t) for t in range(T)])
    return(factor * (cos_term**2 + sin_term**2))

def omega(M, b, x_t, lambdajs):
    '''
    singled out sigma divided by 2pi
    '''
    Ixs = np.array([Ix(x_t = x_t, lambda_j = lam) for lam in lambdajs])
    deltas = np.array([transfer(b = b, lambda_j = lam) for lam in lambdajs])
    sum_fractions = np.sum(Ixs/deltas)
    return(1/M*sum_fractions)
```

Note that since *Omega* as well as I_m in general contain the sum of I_x over all λ_j we will define a numpy array that contains our λ_j defined as:

$$\lambda_j = \frac{2\pi j}{T} \text{ for all j} = 0, 1, 2...$$

If λ_j is uneven we use $\lfloor \lambda_j \rfloor$.
In the code presented above we create an array containing all $I_x(\lambda_j)$ for all elements in the λ-array. We do the same for all and divide the array through each other, which is an element-wise operation when using numpy arrays. Summing up all elements of this array containing all fractions and dividing by M we end up with Ω.
The next code chunk shows our objective function $I_m(b)$ as a function of a single parameter b. We split it up into three parts: the first is the transfer function, or Δ, which we calculate for all λ_j. The second term is the function Ω at the given value of b. Note that we calculate this a priori in the optimization process for all b on our grid and therefore only supply a single element.

```
def I_m(x_t, b, lambdajs, omega):
    '''
    Objective function as discussed in class, log part split up
    '''
    first = transfer(b, lambdajs)
    second = omega
    third = np.array([Ix(x_t = x_t, lambda_j = lam) for lam in lambdajs])
    val = - np.log(first) - np.log(second) - third/(second*first)
    summation = np.sum(val)
    return(summation)
```

Lastly, our optimization function, where we calculate all our objective function I_m for all values of b on our grid and then find the highest value. The index of the highest value is retrieved such that we can find the corresponding b from our grid. Moreover, we retrieve the respective value of σ^2 associated with the optimal b. Note that we need to multiply the value that is returned from our array "sigma" by 2π since the sigma array contains the values of our Ω function for all possible b.

```
def optimization(x_t, bs, lambdajs, M):
    '''
    Calculate all I_m for a grid of points b and find associated value in bs for
        max(I_m)
    '''
    sigmas = np.array([omega(M, param, x_t, lambdajs) for param in bs])
    Ims = np.array([I_m(x_t = x_t, b = param, lambdajs = lambdajs, sigma = sig)
        for param, sig in zip(bs, sigmas)])
    max_i = np.where(Ims == max(Ims))
    param_hat = bs[max_i]
    variance_hat = sigmas[max_i] * 2 * np.pi
    return(param_hat, variance_hat)
```

Now we are ready to implement the simulation study.

```
T = 1000 #set periods
M = int((T-1)/2) #set M
b = 0.25 #true coefficient value
lambdas = np.array([(2*np.pi*j)/T for j in range(M)]) #create array of lambda_j
bs = np.arange(-0.9, 0.9, 0.05)
n_specs = 3 #define number of different specification for epsilon
S = 1000 #define number of iterations
#create empty matrices for DGP
eps = np.empty((n_specs, T+1)) #array 3 x 1001, i x 1001 contains epsilons for
    ith spec.
#preparing empty arrays for estimates
params_hat = np.empty((n_specs, S)) #an empty array in which we save respective
    estimate of each iteration
variances_hat = np.empty((n_specs, S)) #same for the sigma^2 estimation

start = time.time() #time running time

for s in range(S):
    #generate epsilons and x
    eps[0] = rd.normal(0, 1, T+1)
    eps[1] = t.rvs(df = 5, size = T+1)
    eps[2] = rd.uniform(0, 1, T+1)
    eps_lag = eps[:, :T]     #each array in array of arrays eps is taken up to T,
        so last obs is dropped
    x = eps[:, 1:] + b * eps_lag #take only elements i = 1 to i = T-1 of epsilon
        arrays
    for i, k in enumerate(x):
        results = np.array(optimization(k, bs, lambdas, M))
        params_hat[i][s] = results[0] #first element returned by optimization
            function is estimate for b
        variances_hat[i][s] = results[1] #second element is estimate of variance
    print(s)
end = time.time() #end timer
print(end-start)
```

The array "params_hat" and "variances_hat" are 3×1000 arrays, where element i, j is the estimate for specification i in repetition j. Thus, we can now easily apply our bias and MSE functions (see Appendix) to find the bias and MSE for the respective estimates of each specification.

	b	σ_ϵ^2
Bias model I	-2.58	1.9788
Bias model II	-0.917	-0.77
Bias model III	2.083	-0.295
MSE model I	1.979	4.668547
MSE model II	2.108	32.43751
MSE model III	2.216	0.02183261

Table 1: Caption

```
bias_params = np.empty(n_specs) #empty arrays where ith element will be bias of b
    for ith specification
bias_variance = np.empty(n_specs) #same as for b above
true_variances = np.array((1, 5/3, 1/12)) #define true variances (per hand since
    easier)

for i, param in enumerate(params_hat):
    bias_params[i] = bias(param, b, T) #apply bias function to each array
        contained in "params_hat"

for i, param, true in zip(range(n_specs), variances_hat, true_variances):
    bias_variance[i] = bias(param, true, T) #apply bias function to each array
        contained in "variance_hat"

#!calculating MSEs
#applying same principle as above to get the MSEs, need to supply parameter
    estimates as well to MSE function to calculate the variance

param_hat_MSE = np.empty(n_specs)
for i, bias, param in zip(range(n_specs), bias_params, params_hat):
    param_hat_MSE[i] = MSE(bias, param, T)

variance_hat_MSE = np.empty(n_specs)
for i, bias, param in zip(range(n_specs), bias_variance, variances_hat):
    variance_hat_MSE[i] = MSE(bias, param, T)
print(variance_hat_MSE)
```

Our results are summarized in Table 1.

Q16

Important Note: The plots are not entirely correct because I accidentally only generated 1000 y, however, the code for generating 3000 y takes so long that I wasn't able to reproduce correct graphs before the deadline.
To generate the given data process we use the formulas discussed in class, namely

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

Since we cannot integrate up to ∞ we need to approximate ψ_j in the following way:

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

As described in the assignment we take the sum of $M+T$ and then discard the first M observations of our process. However, looking at the anti-derivative of the function $f(x,z)$ we can see that when $d=0$ we would need to divide by zero, which is not possible. Thus, we start at a slightly higher value for our program to be able to distinguish it from zero, which is 0.001.

```
#defining inner function of gamma function, f(x,z)
def f(x, z):
    val = x**(z-1)*np.exp(-x)
    return(val)
#defining gamma function using intergrating function supplied by scipy module
def intf(z):
    if z!= 0:
        val = quad(f, 0, np.inf, args = z)[0]
    else:
        val = quad(f, 0.001, np.inf, args = z)[0]
    return(val)
#defining psi(j, d)
def psi(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)
```

The data generating process uses the below depicted function to generate our y_t of interest:

```
def get_data(length, d_list, epsilon):
    #creating an empty array where data will be saved
    #array contains 5 x length elements, where i x length is data for process
        with d = d_list[i]
    y = np.empty((len(d_list), length))
    for i in range(len(d_list)):
    #looping over all possible values of i
        result = np.empty(length)
        for t in range(length):
            #applying data generating process
            result[t] = np.sum([psi(j, d_list[i]) * epsilon[t-j] for j in range(1,
                t)])
            print(t)
        y[i, :] = result
    return(y)
```

As we can now generate data and have all necessary functions in place we can calculate the autocovariances following the slides:

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

The autocorrelations $\rho(h)$ are then retireved by $\rho(h) = \frac{\gamma(h)}{\gamma(0)}$. We define the respective functions below:

```
def get_gamma0(data, d_list, d):
    '''
    data must be an array
    '''
    pos = np.where(d_list == d)
    sigma2 = np.var(data[pos, :])
    upper = intf(1 - 2*d)
    lower = intf(1-d)**2
    final = sigma2*(upper/lower)
    return(final)

def get_corrs(data, laglength, d_list):
    autocorrs = np.empty((len(d_list), laglength + 1))
    estimated = np.empty((len(d_list), laglength + 1))
    for i in range(len(d_list)):
        lags = np.empty((len(d_list), laglength+1))
        gamma_0 = get_gamma0(data, d_list, d_list[i])
        lags[i][0] = gamma_0
```

```
        for h in range(1, laglength + 1):
            factor = (h-1+d_list[i])/(h-d_list[i])
            gammah = factor * lags[i][h-1]
            lags[i][h] = gammah
        #getting autocorrs from autocovs
        autocorrs[i, :] = lags[i, :]/lags[i, 0]
        estimated[i, :] = acovf(data[i, :], demean = False, nlag = laglength)
    final = np.array((autocorrs, estimated))
    return(final)
```

Note that we use the function "acovf" provided by the statsmodels module to calculate the estimated autocovariances of the process.

Figure (1) and Figure (2) shows the plotted approximated real autocorrelations and their correspoding estimates for $d \in \{-0.45, -0.25\}$ and for $d \in \{0.25, 0.45\}$ respectively. Figure (3) shows the special case of $d = 0$, where autocovariances are zero for $h > 0$. Figure (4) shows some example process $\{y_t\}_{t=1}^{1000}$.

Appendix

```
def bias(estimate, true_val, T):
    '''
    Calculate bias as given in assignment.
    '''
    reps = len(estimate)
    summation = np.sum(estimate)
    bias = 1/reps * summation - true_val
    return(bias)
def MSE(bias, parameter, T):
    '''
    *parameter = parameter estimates
    '''
    bias2 = bias**2
    variance = np.var(parameter)
    return((bias2 + variance)*T)
```

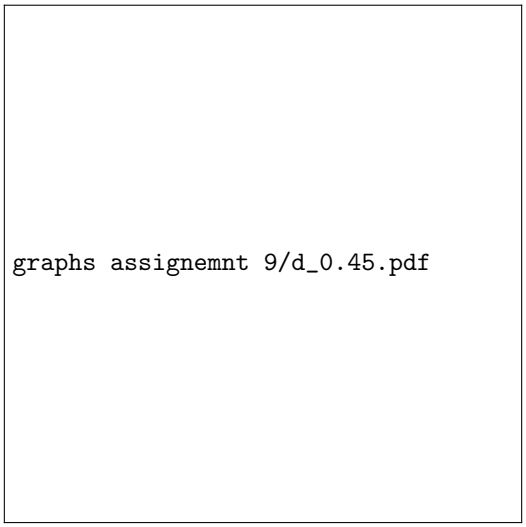

Figure 1: d = 0.45

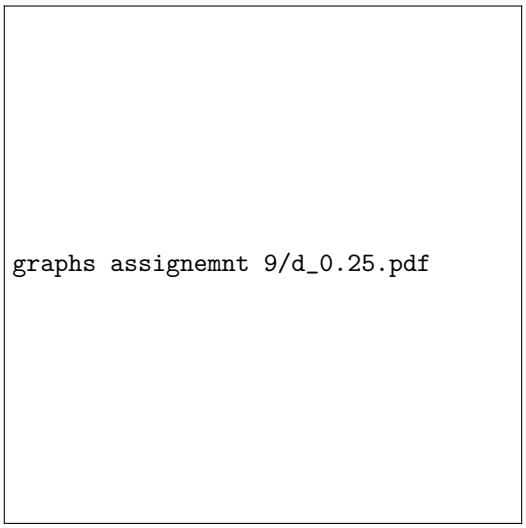

Figure 2: d = 0.45

Figure 3: Figures for positive d

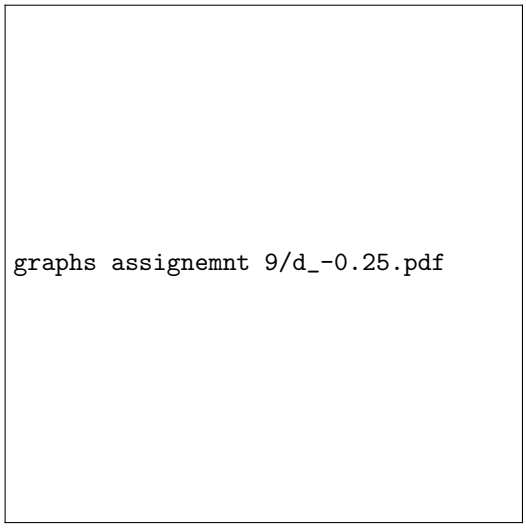

Figure 4

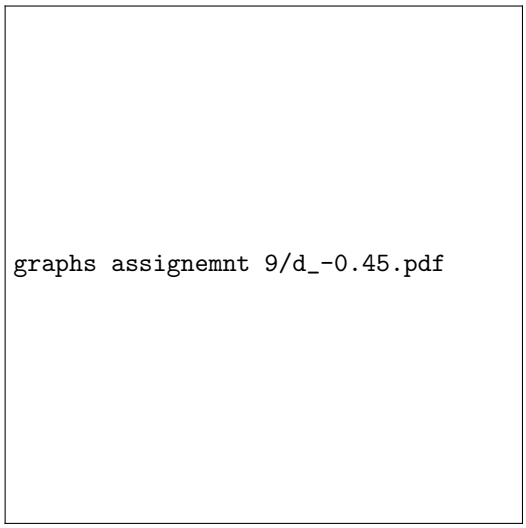

Figure 5

Figure 6: Figures for negative d