Time Series coursework

Jonathan Cheung

December 2020

1 Question 1

The following code is all in Python. We are using the time series "http://www2.imperial.ac.uk/ eakc07/time_series/29.csv" which we call X29. First, we create a function 'S_AR(f,phis,sigma2)' that will evaluate the parametric form of the spectral density function for an AR(p) process. This functions uses the result:

$S_X(f) = \frac{\sigma_\epsilon^2}{|1 - \phi_{1,p}e^{-i2\pi f} - ... - \phi_{p,p}e^{-i2\pi fp}|}$

from page 50 in the notes.

```
import numpy as np #importing our packages
import pandas as pd
from scipy import linalg
import matplotlib.pyplot as plt

def S_AR(f,phis,sigma2):
    """
    evaluates parametric form of the spectral density function
    for an AR(p) process on a designated set of frequencies.
    Parameters
    f: vector of frequencies at which to evaluate the spectral density function
    phis: vector [phi_{1,p} ,..., phi_{p,p}]
    sigma2: variance of white noise
    Returns:
    S:vector of values of spectral density function evaluated at elements of f
    """
    #len_f = len(f)
    len_phi = len(phis) #check length of phis

    one_to_p = np.arange(1,len_phi+1,1)
    #matrix where each row element is f*n for n equals 1 to p
    indices = np.outer(f,one_to_p)
    #elt. exp of indices matrix multiplied by -i*2pi*f*n
    poly_exp = np.exp(1j * 2 * np.pi * indices)
    #multiply row of the poly_exp matrix with the phi vector
    G_phi_f = np.dot(poly_exp,phis)

    sdf = sigma2/(np.abs(1-G_phi_f)**2)

    return sdf
```

We also write a function 'AR2_sim' that will generate a burned in sequence of an AR(2) process. We remind ourselves that an AR(2) process has form $X_t - \phi_{1,2}X_{t-1} - \phi_{2,2}X_{t-2} = \epsilon_t$

```
def AR2_sim(phis,sigma2,N):
    """
    Parameters
    ----------
    phis : phi vector
    sigma2 : variance of white noise
    N : Length of sequence to create

    Returns
    -------
    burnin : burned in sequence vector
    """
    X_t= np.zeros(N+100) #preallocate space
    #generate white noise process
    epsi = np.random.normal(0,sigma2,size=N-2+100)

    for i in range(N-2+100): #generate sequence
        X_t[i+2] =  phis[0]*X_t[i+1] +phis[1]*X_t[i] +epsi[i]
    burnin = X_t[100:] #exclude the first 100 values

    return burnin
```

We create a function 'acvs_hat' that will estimate the autocovariance of a time series at different lags. This uses the result $\hat{s}_\tau^{(p)} = \frac{1}{N}\sum_{t=1}^{N-|\tau|}(X_t - \bar{X})(X_{t+|\tau|} - \bar{X})$ where τ is an integer of magnitude less than N from page 56 in the notes.

```
def acvs_hat(X,tau):
    """
    estimates acvs for X at tau values
    Parameters
    ----------
    X : Time series vector
    tau : Vector of lags
    Returns
    -------
    s_hat : vector of acvs estimates at each tau
    """
    N = len(X) #length of time series
    #preallocate space
    s_hat = np.zeros_like(tau,dtype=float)
    #dtype float otherwise rounding error
    for i in range(len(tau)): #need an estimate for each tau
        #creating some indexes so we can vectorise the sum
        end_t = N - np.abs(tau[i])
        start_t = np.abs(tau[i])
        s_hat[i] = (1/N)* np.dot(X[:end_t],X[start_t:])

    return s_hat
```

2 Question 2

We will now explore the bias properties for the periodogram and direct spectral estimators for AR processes.

Here is a function for computing the periodogram. It works by applying the fast fourier transform onto the time series data. This works due to the result $\hat{S}^{(p)}(f) = \frac{1}{N}|\sum_{t=1}^{N} X_t e^{-i2\pi ft}|^2$ from page 58.

```
def periodogram(X):
    """
    computes periodogram  of X at fourier frequencies
    Parameters
    ----------
    X : vector time series
    Returns
    -------
    p : periodogram estimate of sdf
    """
    N = len(X)
    #fourier transform of Xts is periodogram
    p = (1/N)* (np.abs(np.fft.fft(X)))**2
    return p
```

We also have a function for the direct spectral estimate. This uses the Hanning taper:
$h_t = \frac{1}{2}[\frac{8}{3(N+1)}]^{1/2}[1 - cos(\frac{2\pi t}{N+1})], t = 1, ..., N$ (Where N is the length of our time series)
We are using the result $\hat{S}^{(d)}(f) = |\sum_{t=1}^{N} h_t X_t e^{-i2\pi ft}|^2$ from page 62.

```
def direct(X):
    """
    computes direct spectral estimate of X at fourier freq.
    Parameters
    ----------
    X : vector time series
    Returns
    -------
    sdf_hat : estimate of sdf
    """
    N=len(X)
    t= np.arange(1,N+1,1)
    #create hanning taper
    h_t = 0.5*np.sqrt((8/(3*(N+1))))*(1-np.cos(2*np.pi*t/(N+1)))
    htxt = np.multiply(h_t,X) #apply taper
    Jf = np.fft.fft(htxt) #apply fft
    sdf_hat = (np.abs(Jf))**2 #estimate sdf
    return sdf_hat
```

We will use our function 'AR2_sim' to simulate 10000 realisations of an AR(2) process with varying lengths, f'=1/8, r=0.95, $\sigma_\epsilon^2 = 1$, and complex conjugate roots: $\frac{1}{r}e^{i2\pi f'}$ and $\frac{1}{r}e^{-i2\pi f'}$. Our AR(2) process can then be writen as $X_t - 2rcos(2\pi f')X_{t-1} + r^2 X_{t-2}$ (page 51 of lecture notes) and so our phi values can be seen to be as $2rcos(2\pi f')$ and $-r^2$.

The code below is a function that takes N as its input and will simulate 10000 realisations of length N with the above conditions. It will then output two arrays 'mean_bias_p' and 'mean_bias_d'. These arrays

each contain the average empirical bias for the 10000 realisations at the frequencies 1/8, 2/8, and 3/8. The first array corresponds to the periodogram and the second array corresponds to the direct spectral estimate. We calculate bias by using our 'S_AR' function to calculate the true spectral density function and then taking the difference between this and our estimated values.

```
def emp_bias(N,r=0.95,fdash=1/8,sigma2=1,sim=10000):

    bias_p = np.zeros((sim,3)) #preallocate space
    bias_d = np.zeros((sim,3)) #preallocate space

    one_i = int((1/8) * (N)) #index for 1/8 frequency
    two_i = int((2/8) * (N)) #2/8 index
    three_i=int((3/8) * (N)) #3/8 index
    #calculate phis using the given roots
    phis=[2*r*np.cos(2*np.pi*fdash),-r**2]
    f=[1/8,2/8,3/8] #frequencies to evaluate
    real_sdf = S_AR(f, phis,sigma2) #true sdf values at the given frequencies
    for i in range(sim): #10000 simulations
        X_sim = AR2_sim(phis,sigma2,N) #simulate one run of length N

        prd = periodogram(X_sim) #periodogram estimate
        dse = direct(X_sim) #direct spectral estimate
        #appending sdf values to an array
        bias_p[i,0] = prd[one_i]
        bias_p[i,1] = prd[two_i]
        bias_p[i,2] = prd[three_i]
        #appending for the direct method
        bias_d[i,0] = dse[one_i]
        bias_d[i,1] = dse[two_i]
        bias_d[i,2] = dse[three_i]
    #taking the mean of the sdf values and then the difference
    mean_bias_p = np.mean(bias_p,axis=0) -real_sdf
    mean_bias_d = np.mean(bias_d,axis=0) -real_sdf

    return mean_bias_p, mean_bias_d
```

If we set N=16, we get the following output. Our immediate observations are that the bias at 1/8 is a lot larger than at 2/8 or 3/8. We also have that the bias for direct is better at the higher frequencies 2/8 and 3/8 but worse at 1/8.

frequency	1/8	2/8	3/8
periodogram	-142.729	0.74129	0.28116
direct	-158.394	0.26823	0.00857

We now explore what happens if we vary our N value. The code below will evaluate the function above at N=16,32,64,...4096. Then it will plot three separate graphs (fig 1a,1b,1c). Each graph corresponds to one of the frequencies (1/8, 2/8, 3/8) and will display the relationship between bias and sequence length for each of the two methods. It will also plot the true spectral density function (fig 1d).

```
(N,r,fdash,sigma20)=(16,0.95,1/8,1)
f = np.linspace(0,1,1000)
#plot the true sdf
plot_sdf = sigma2/((1-2*r*np.cos(2*np.pi*(fdash+f))+r**2)\
(1-2*r*np.cos(2*np.pi*(fdash-f))+r**2))
plt.figure()
plt.plot(plot_sdf)
plt.title('true spectral density function')
plt.ylabel('S(f)')
plt.xlabel('frequency f')
plt.xticks(np.linspace(0,1000,8), (1/8) *np.arange(0,8,1))
#preallocate space for bias values
plot1 = np.zeros((9,2))
plot2 = np.zeros((9,2))
plot3 = np.zeros((9,2))
#loop through 16 to 4096
for i in np.arange(4,13,1):
    print(i) #shows the progress
    (p_bias,d_bias) = emp_bias(2**i) #calculate bias
    plot1[i-4,:] = np.array([p_bias[0],d_bias[0]])
    plot2[i-4,:] = np.array([p_bias[1],d_bias[1]])
    plot3[i-4,:] = np.array([p_bias[2],d_bias[2]])

plt.figure() #bias at 1/8
lineObjects = plt.plot(plot1)
plt.title('empirical bias against sequence length at f=1/8')
plt.ylabel('empirical bias')
plt.xlabel('length of simulated sequence (log2 scale)')
plt.xticks(np.arange(plot1.shape[0]), 2**np.arange(4, plot1.shape[0]+4))
plt.legend(lineObjects, ('Periodogram','Direct'))

plt.figure() #bias at 2/8
lineObjects = plt.plot(plot2)
plt.title('empirical bias against sequence length at f=2/8')
plt.ylabel('empirical bias')
plt.xlabel('length of simulated sequence (log2 scale)')
plt.xticks(np.arange(plot2.shape[0]), 2**np.arange(4, plot2.shape[0]+4))
plt.legend(lineObjects, ('Periodogram','Direct'))

plt.figure() #bias at 3/8
lineObjects = plt.plot(plot3)
plt.title('empirical bias against sequence length at 3/8')
plt.ylabel('empirical bias')
plt.xlabel('length of simulated sequence (log2 scale)')
plt.xticks(np.arange(plot3.shape[0]), 2**np.arange(4, plot3.shape[0]+4))
```

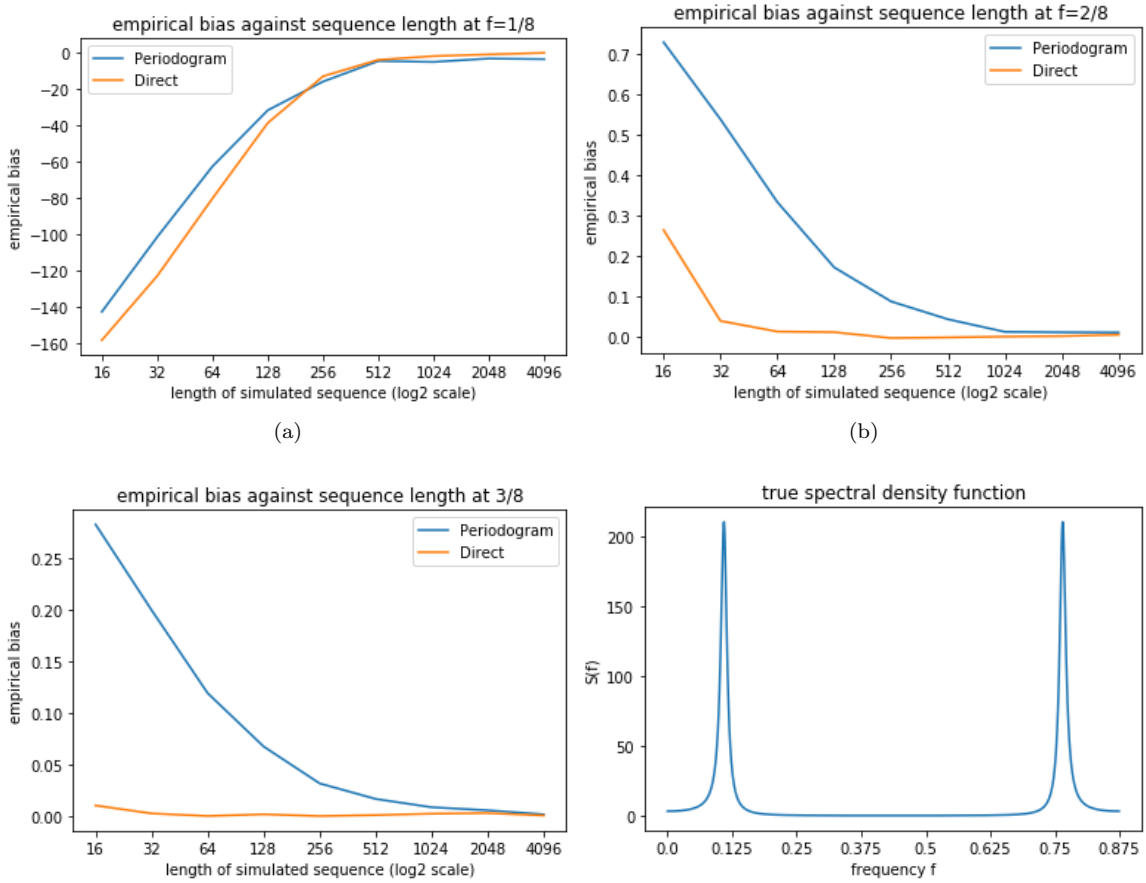


Figure 1: empirical bias at f=1/8, 2/8, 3/8 and the true sdf

We see that the bias decreases as the length of the sequence (N) increases. This makes sense since the more data we have, the easier it is to fit a model.

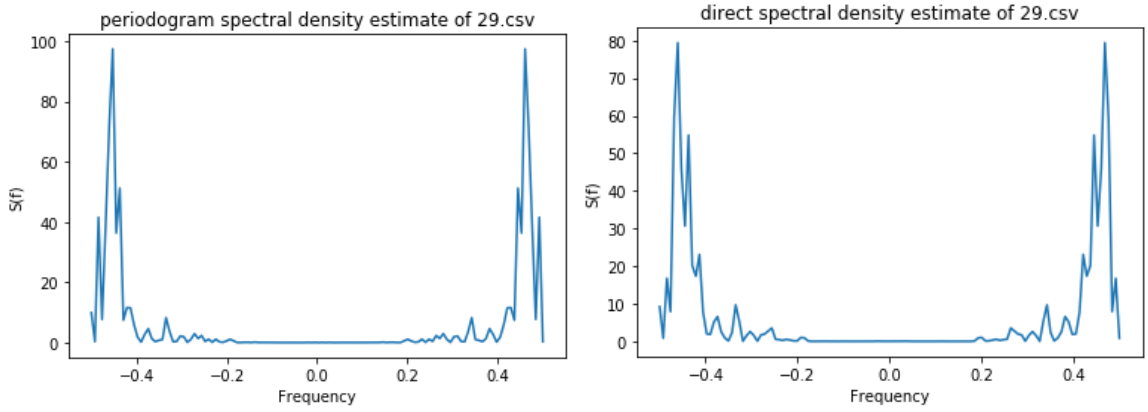
We also note that the direct spectral estimate generally has less bias than the periodogram method, especially at higher frequencies. This agrees with what we learned in the course; applying a taper helps with side-lobe leakage which happens at higher frequencies. Though the periodogram does perform better for small N (up to 128) for f=1/8

Lastly, the empirical bias is higher at 1/8 (fig 1a) for both methods than at other frequencies. This can be explained by referring to the plot of the true spectral density function (fig 1d). There is a huge spike around the 1/8 frequency, which is hard to fit. Whereas at 2/8 and 3/8, it is a lot flatter.

3 Question 3

The functions below will plot the shifted periodogram and direct estimate (fig 2). The effect of the taper can be seen as the tails of the time series are lower in the direct graph.

```
def periodogramshift(X):
    N = len(X)
    #fourier transform of Xts is periodogram
    p = (1/N)* (np.abs(np.fft.fft(X)))**2
    p = np.fft.fftshift(p) #shift
    x = np.linspace(-0.5,0.5,num=N)
    plt.plot(x,p)
    plt.title('periodogram spectral density estimate of 29.csv')
    plt.ylabel('S(f)')
    plt.xlabel('Frequency')
    return p

def directshift(X):
    N=len(X)
    x = np.linspace(-0.5,0.5,num=N)
    t= np.arange(1,N+1,1)
    #create hanning taper
    h_t = 0.5*np.sqrt((8/(3*(N+1))))*(1-np.cos(2*np.pi*t/(N+1)))
    htxt = np.multiply(h_t,X) #apply taper
    Jf = np.fft.fft(htxt) #estimate sdf
    Jf = np.fft.fftshift(Jf) #shift
    sdf_hat = (np.abs(Jf))**2
    plt.plot(x, sdf_hat)
    plt.title('direct spectral density estimate of 29.csv')
    plt.ylabel('S(f)')
    plt.xlabel('Frequency')
    return sdf_hat
```



(a) (b)

Figure 2: spectral density estimates

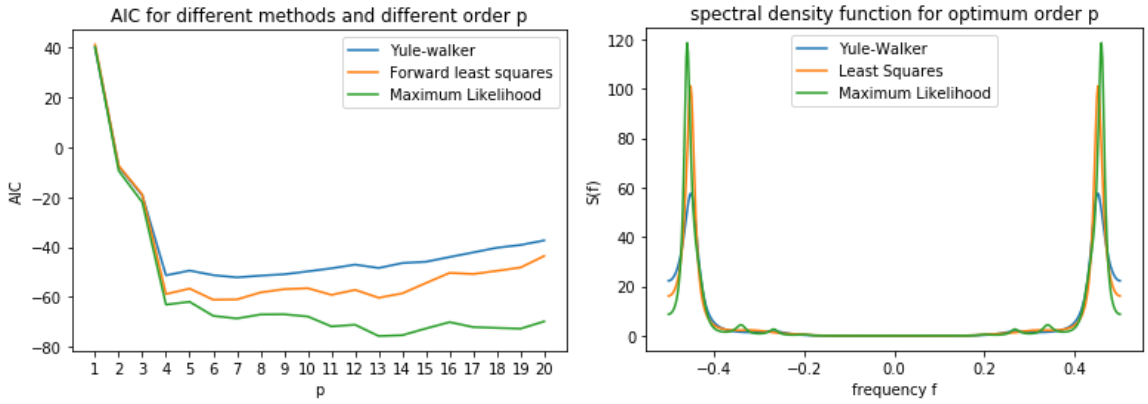
We also have the following code that will fit an AR(p) model with different methods

```
def yw(p,X):
    """
    yule walker estimate for phi and sigma2 parameters
    Parameters
    ----------
    p : degree of AR(p) model you want to fit
    X : time series data
    Returns
    -------
    phis : vector containing estimates of the phis
        for phi_1,p ... to... phi_p,p
    sigma2 : estimate of variance for 0 mean white noise process
    """
    s_hat = acvs_hat(X,np.arange(0,p+1,1))
    #solve Ax=Id, where A is the toeplitz matrix, so x is inverse
    big_gam_inv = linalg.solve_toeplitz((s_hat[:-1],s_hat[:-1]), np.eye(p))
    small_gamma = s_hat[1:] #ignoring 0 entry
    phis = np.dot(big_gam_inv,small_gamma)
    sigma2 = s_hat[0] - np.dot(phis,s_hat[1:])
    return phis,sigma2

def fls(p,X):
    """
    fwd least squares estimator for phi and sigma2 parameters inputs/outputs like above
    """
    Fmat = linalg.toeplitz(X[p-1:-1],X[p-1::-1]) #we want X_p term on top left corner
    ftf_inv_ft = linalg.inv(np.transpose(Fmat)@Fmat) @ np.transpose(Fmat)
    phi_hat = np.dot(ftf_inv_ft,X[p:])
    N=len(X)
    bottom = N-2*p
    top1 = np.transpose(X[p:] - Fmat@phi_hat)
    top2 = X[p:]- Fmat@phi_hat
    sigma2 = top1@top2/bottom
    return phi_hat, sigma2

def maxlh(p,X):
    """
    max likelihood estimator for phi and sigma2 parameters, inputs/outputs like above
    """
    Fmat = linalg.toeplitz(X[p-1:-1],X[p-1::-1])
    ftf_inv_ft = linalg.inv(np.transpose(Fmat)@Fmat) @ np.transpose(Fmat)
    phi_hat = np.dot(ftf_inv_ft,X[p:])
    N=len(X)
    sigma2 = (np.transpose(X[p:] - Fmat@phi_hat) @ (X[p:] - Fmat@phi_hat))/(N-p)
    #only difference to fls
    return phi_hat, sigma2
```

The code below will take as its input, our time series X and return a table of aic values and a plot of these values. The first column corresponds to the Yule-Walker, the second to the least squares, and the third to the maximum likelihood. From top to bottom, the rows correspond to increasing p value from 1 to 20.

```
def aic_table(X,pcount=20):
    N=len(X)
    AIC = np.zeros((20,3))
    for i in range (pcount):
        p=i+1
        yw_sig = yw(p,X)[1]
        ls_sig = fls(p,X)[1]
        ml_sig = maxlh(p,X)[1]
        AIC_yw = 2*p + (N*np.log(yw_sig)) #calculate aic values
        AIC_ls = 2*p +N*np.log(ls_sig)
        AIC_ml = 2*p +N*np.log(ml_sig)
        AIC[i,:]=[AIC_yw,AIC_ls,AIC_ml] #append aic values
    plt.figure() #plot figure
    lineObjects = plt.plot(AIC)
    plt.legend(lineObjects, ('Yule-walker','Forward least squares','Maximum Likelihood'))
    plt.xticks(np.arange(AIC.shape[0]), np.arange(1, AIC.shape[0]+1))
    plt.ylabel('AIC')
    plt.xlabel('p')
    plt.title('AIC for different methods and different order p')
    return AIC
```

We can look to table 1 to identify the lowest AIC value. For the Yule-Walker, it is at p=4 the AIC is at its lowest. For the least squares method, p=6 has the lowest AIC. In the maximum likelihood case, the AIC is lowest at p=13. However, in all three cases, the AIC stops decreasing so significantly after p=4. We can rerun our function with these p values and our time series X29 to return a vector of phis $(\phi_{1,p}, ..., \phi_{p,p})$ and a σ_ϵ^2 value. The code below will also plot the three associated spectral density functions (fig 3b).



(a) (b)

Figure 3: exploring the optimum choice of p for the three methods

```
In [1]:yw(4,X29)
Out[1]:
(array([-1.73417234, -1.50012426, -1.08104074, -0.48308769]),
 0.6290593482098412)

In [2] fls(6,X29)
Out[2]:
```

```
(array([-1.87734151, -1.9244798 , -1.69349978, -1.14382319, -0.55939292,
        -0.24895589]), 0.5648159283956401)

In [3] maxlh(13,X29)
Out[3]:
(array([-1.90203653, -1.86273716, -1.37425528, -0.54485627,  0.20477167,
         0.46557956,  0.48534969,  0.20362107,  0.04324895, -0.02541585,
         0.07174687,  0.26561352,  0.20042907]), 0.4518875150491666)

f=np.linspace(-0.5,0.5,512) #x axis
yplot = S_AR(f,yw(4,X29)[0],yw(4,X29)[1]) #y axis
lplot = S_AR(f,fls(6,X29)[0],fls(6,X29)[1])
mplot = S_AR(f,maxlh(13,X29)[0],maxlh(13,X29)[1])
plt.figure #plot
plt.plot(f,yplot,label='Yule-Walker')
plt.plot(f,lplot,label='Least Squares')
plt.plot(f,mplot,label='Maximum Likelihood')
plt.xlabel('frequency f')
plt.ylabel('S(f)')
plt.legend()
plt.title('spectral density function for optimum order p')
plt.show
```

p	yule-walker	least squares	max-likelihood
1	39.8957409	41.14090298	40.12904001
2	-7.72838038	-7.39368328	-9.44172697
3	-19.31505138	-18.98352667	-22.09299132
4	-51.3317982	-58.8533372	-63.05043452
5	-49.42722796	-56.63445518	-61.94642074
6	-51.2585376	-61.12069009	-67.57583936
7	-52.1572264	-61.00026966	-68.6280581
8	-51.4484946	-58.18676803	-67.01785558
9	-50.9061192	-56.88101299	-66.94733329
10	-49.74434442	-56.49403222	-67.82886708
11	-48.52464435	-59.16747443	-71.80553404
12	-47.03711233	-57.15311886	-71.13062823
13	-48.39449012	-60.31842247	-75.6732148
14	-46.40739076	-58.55118167	-75.32279925
15	-45.88974654	-54.39082186	-72.62062538
16	-43.99999361	-50.35110694	-70.08239396
17	-42.14861626	-50.80453292	-72.08266656
18	-40.26988685	-49.53296419	-72.40551315
19	-39.17376753	-48.19276661	-72.70965773
20	-37.34975668	-43.57687443	-69.79055924

Table 1: AIC values for varying p and different methods

4 Question 4

Using these parameters, future terms can be forecasted. This relies on our assumption that future innovations $\epsilon_t = 0$. We can forecast one step at a time using the equation: $X_t(l) = \phi_{1,p}X_T + ... + \phi_{p,p}X_{t-p+1}$ (page 92).

The following code will implement this for our time series X29. It will use the first 118 values to forecast the final 10 values. All three methods will be used with the parameters that we decided were optimal (with AIC) earlier. Finally, setting the variables 'METHOD_plot' (where METHOD can be yw, fls, ml) to 1 will allow us to plot the real time series against these selected methods.

```
def forecast(X29,yw_plot,fls_plot,ml_plot):
    (yphis,ysig) = yw(4,X29) #return and save the parameters
    (lphis,lsig) = fls(6,X29)
    (mphis,msig) = maxlh(13,X29)

    past = X29[0:118] #preallocate space to put new forecasts
    ywforecast = np.concatenate((past,np.zeros(10)))
    lsforecast = np.concatenate((past,np.zeros(10)))
    mlforecast = np.concatenate((past,np.zeros(10)))

    for i in np.arange(118,128,1): #forecast for t=119 to 128
        ywforecast[i] = np.dot(yphis,ywforecast[i-1:i-1-4:-1])
        lsforecast[i] = np.dot(lphis,ywforecast[i-1:i-1-6:-1])
        mlforecast[i] = np.dot(mphis,ywforecast[i-1:i-1-13:-1])

    t= np.arange(110,129,1)
    plt.figure #plot
    if yw_plot==1: #set to 1 if we want to plot the yw forecast
       plt.plot(t,ywforecast[109:],label='Yule-Walker')
    if fls_plot==1:
        plt.plot(t,lsforecast[109:],label='least Squares')
    if ml_plot==1:
        plt.plot(t,mlforecast[109:],label='Maximum Likelihood')
    plt.plot(t,X29[109:],label='real data')
    plt.title('forecasted time series vs real time series')
    plt.xlabel('time')
    plt.legend()
    plt.show

    return ywforecast[118:],lsforecast[118:],mlforecast[118:]
```

If we use our function with the inputs below, we will return the forecasted values (table 2) and plots to compare these values (fig 4).

```
In [4] forecast(X29,1,0,0)

In [5] forecast(X29,0,1,0)

In [6] forecast(X29,0,0,1)

In [7] forecast(X29,1,1,1)
```

Time	119	120	121	122	123
True values	-4.2072	3.9478	-3.2767	2.4216	-2.0798
Yule Walker	-4.058382	3.21771367	-2.62124913	2.02143202	-1.09123729
Forward Least Squares	-4.08061289	3.39412432	-2.5984333	2.04016659	-1.18000839
Maximum Likelihood	-4.30828543	3.26741007	-2.80840438	2.16587797	-1.24902095
Time	124	125	126	127	128
True values	1.2527	-1.3185	1.694	-1.0528	-0.68732
Yule Walker	0.13923357	0.47657934	-0.83219531,	1.104888	-1.25013354
Forward Least Squares	0.11296989	0.62401789	-0.96156564	1.17933864	-1.33187163
Maximum Likelihood	0.15250045	0.4524347	-0.82189643	1.1238235	-1.43388287

Table 2: forecasted values against real values

All three methods seem to perform similarly despite having different order p. There are time steps where each of the three methods appear to perform better than the other two. The methods also appear to be reasonably accurate with the first four values. After that they are not so close to the true values. This is probably because the effect of white noise and that we are predicting future values with future values.

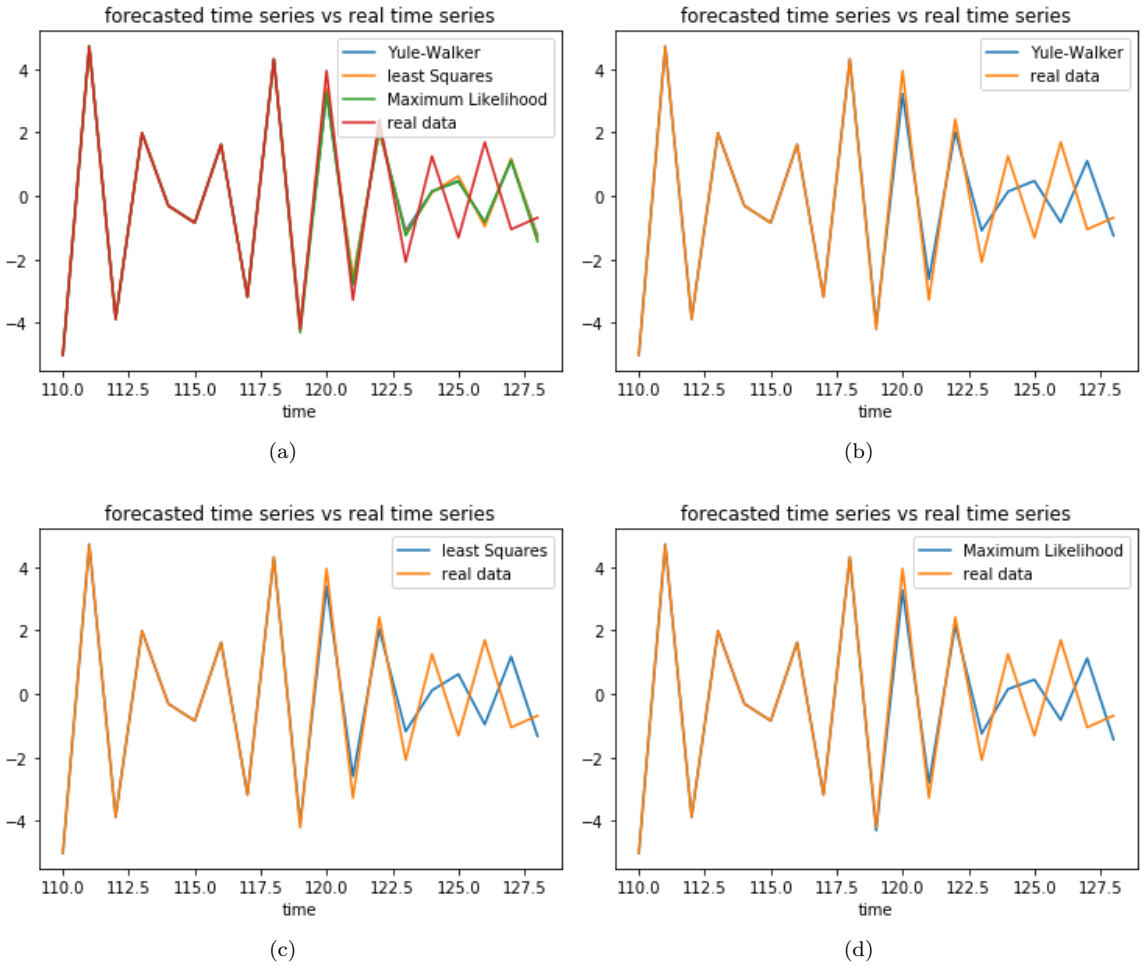


Figure 4: visualising the difference between the forecasted values