CW

December 15, 2021

```
[ ]: import numpy as np
     import pandas as pd
     from scipy.fft import fft

     import matplotlib.pyplot as plt
     from matplotlib import cm

     plt.style.use('seaborn-white')
```

0.1 1

0.1.1 (a)

```
[ ]: def S_ARMA(f, sigma2, phis=[], thetas=[]):
         '''
         Computes the theoretical sdf for an ARMA(p, q) process.

         param f: numpy array of frequencies at which sdf should be evaluated
         param sigma2: scalar; the variance of the white noise process
         param phis: numpy array containing the phi parameters
         param thetas: numpy array containing the theta parameters

         return: numpy array of sdf evaluated at f
         '''
         p, q = len(phis), len(thetas)
         xis_t = np.exp(-1j*2*np.pi*np.outer(f, np.arange(1, q+1)))
         xis_p = np.exp(-1j*2*np.pi*np.outer(f, np.arange(1, p+1)))
         G_t = 1 - (thetas * xis_t).sum(1)
         G_p = 1 - (phis * xis_p).sum(1)
         return (sigma2 * np.abs(G_t)**2 / np.abs(G_p)**2)
```

```
[ ]: ## Testing

     f = np.random.randn(100)
     sigma2 = np.random.randn()
     phis = [np.random.randn()]
     np.allclose(
         S_ARMA(f, sigma2, phis=phis),
```

```
    sigma2 / (1 + phis[0]**2 - 2*phis[0]*np.cos(2*np.pi*f)),
)
```

[ ]: True

0.1.2 (b)

```
[ ]: def ARMA22_sim(phis, thetas, sigma2, N):
        """
        Simulates a Gaussian ARMA(2,2) process.

        param phis: numpy array containing the phi parameters
        param thetas: numpy array containing the theta parameters
        param sigma2: scalar; variance of the white noise process
        N: scalar; length of the simulated process

        return: numpy array of size N; realiasation of an ARMA(2,2) process
        """
        eps = np.random.normal(size=100+N, scale=np.sqrt(sigma2))
        X = np.zeros(100 + N)
        for i in range(2, 100+N):
            X[i] = phis[0]*X[i-1] + phis[1]*X[i-2] + eps[i] \
                - (thetas[0]*eps[i-1] + thetas[1]*eps[i-2])
        return X[-N:]
```

0.1.3 (c)

```
[ ]: def periodogram(X):
        """
        Computes the periodogram at the Fourier frequencies \
        for a time series X.
        """
        N = X.shape[-1]
        S = fft(X)
        return np.abs(S)**2 / N


    def tapered_series(X, taper):
        N = X.shape[-1]

        H = np.ones(N,)
        tt = np.floor(taper*N)
        H[:int(tt//2)] = (1 - np.cos(2*np.pi*np.arange(tt//2) / (tt+1))) / 2
        H[N-int(tt//2):] = (1 - np.cos(2*np.pi*(N+1-np.arange(N-int(tt//2), N)) /␣
     ↪(tt+1))) / 2
        H /= np.sqrt((H**2).sum())
```

```
    return H * X

def direct(X, p):
    """
    Computes the direct spectral estimate at the Fourier \
    frequencies using the p x 100% cosine taper for a \
    time series X.
    """
    hX = tapered_series(X, p)
    return np.abs(fft(hX))**2
```

0.1.4 (d)

(A) By result in section 2.3.2 in course notes, an ARMA(2,2) process is stationary if all the roots of

$$\Phi(z) = 1 - \phi_{1,2}z - \phi_{2,2}z^2$$

lie outside the unit circle. Let

$$z_1 = \frac{1}{r}e^{-i2\pi f'}, \quad z_2 = \frac{1}{r}e^{i2\pi f'}$$

be the roots of $\Phi(z)$. By similar arguments in page 51 of course notes, we have that $\phi_{1,2} = r\cos(2\pi f')$, $\phi_{2,2} = -r^2$, the ARMA process can be written as

$$X_t - 2r\cos(2\pi f')X_{t-1} + r^2 X_{t-2} = \epsilon_t - \theta_{1,2}\epsilon_{t-1} - \theta_{2,2}\epsilon_{t-2}.$$

We also have that

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

and

$$|1 - \phi_{1,2}e^{-2\pi f} - \phi_{2,2}e^{-2\pi f}|^2 = (1 - 2r\cos(2\pi(f' + f)) + r^2)(1 - 2r\cos(2\pi(f' - f)) + r^2)$$

[ ]:
```
r = 0.8
N = 128
fp = 12/128
phis = np.array([2*r*np.cos(2*np.pi*fp), -r**2])
thetas = np.array([-0.5, -0.2])

np.random.seed(7)
X = np.array([ARMA22_sim(phis, thetas, 1, N) for _ in range(10000)])
```

[ ]:
```
per_single = periodogram(X).T[[12, 32, 60]]     # periodograms

# cosine tapers
ps = [0.05, 0.1, 0.25, 0.5]
dse_p1_single = direct(X, 0.05).T[[12, 32, 60]]
dse_p2_single = direct(X, 0.1).T[[12, 32, 60]]
dse_p3_single = direct(X, 0.25).T[[12, 32, 60]]
```

```
dse_p4_single = direct(X, 0.5).T[[12, 32, 60]]
```

(B)

```
[ ]: fs = np.array([12/128, 32/128, 60/128])

estimates = np.dstack([per_single, dse_p1_single, dse_p2_single, dse_p3_single,␣
 ↪dse_p4_single])
true_sdf = S_ARMA(fs, sigma2=1, phis=phis, thetas=thetas)
sample_bias_vals = estimates.mean(1).T - true_sdf

# Sample bias table for each estimator when r=0.8
sample_bias = pd.DataFrame(sample_bias_vals, columns=['f=' + str(int(f*128)) +␣
 ↪'/128' for f in fs])
sample_bias.rename_axis(columns='freq', inplace=True)
sample_bias.set_axis(['periodogram'] + [f'{int(p*100)}% cosine taper'  for p in␣
 ↪ps], axis='index', inplace=True)
sample_bias.index.name = 'Sample bias (r=0.8)'
sample_bias
```

```
[ ]: freq                  f=12/128  f=32/128  f=60/128
Sample bias (r=0.8)
periodogram          -2.561065  0.129317  0.051919
5% cosine taper      -2.318748  0.068238  0.005019
10% cosine taper     -2.041909  0.018517  0.000321
25% cosine taper     -1.721829 -0.004065  0.000002
50% cosine taper     -1.513529 -0.007609  0.000101
```

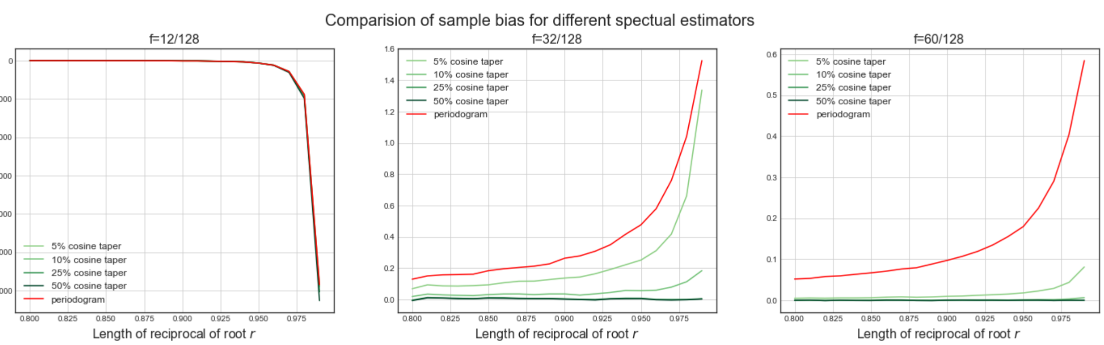
(C)

```
[ ]: N = 128
fp = 12/128
thetas = [-0.5, -0.2]
rs = np.arange(0.8, 1.0, 0.01)
phis = np.array([2*rs*np.cos(2*np.pi*fp), -rs**2]).T   # 20x2 array, recording␣
 ↪phi values for each r

np.random.seed(7)
XX = np.array([[ARMA22_sim(phi, thetas, 1, N) for _ in range(10000)] for phi in␣
 ↪phis])    # shape=(20, 10000, 128)

perXX = periodogram(XX)[..., [12, 32, 60]]     # shape=(20, 10000, 3)

ps = [0.05, 0.1, 0.25, 0.5]
dse_p1 = direct(XX, 0.05)[..., [12, 32, 60]]
dse_p2 = direct(XX, 0.1)[..., [12, 32, 60]]
dse_p3 = direct(XX, 0.25)[..., [12, 32, 60]]
dse_p4 = direct(XX, 0.5)[..., [12, 32, 60]]
```

[ ]:
```python
true_sdf = np.array([S_ARMA(fs, sigma2=1, phis=phi, thetas=thetas) for phi in␣
↪phis])  #  20x3 array

sample_mean = np.stack((perXX, dse_p1, dse_p2, dse_p3, dse_p4)).mean(2)
sample_bias = sample_mean - true_sdf     # size=(5, 20, 3)
```

(D)

[ ]:
```python
fig = plt.figure(figsize=(24, 6))

for i in range(3):
    plt.subplot(1, 3, i+1)
    for ip in range(4):
        plt.plot(rs, sample_bias[ip+1, :, i], c=cm.YlGn(1.25*ps[ip]+0.4),␣
↪label=f'{int(ps[ip]*100)}% cosine taper')
    plt.plot(rs, sample_bias[0, :, i], 'r-', label='periodogram')
    plt.xlabel(r'Length of reciprocal of root $r$', fontsize=16)
    plt.title(f'f={int(fs[i]*128)}/128', fontsize=16)
    plt.grid()
    plt.legend(fontsize=12)
fig.suptitle('Comparision of sample bias for different spectual estimators',␣
↪fontsize=20)
plt.show()
```

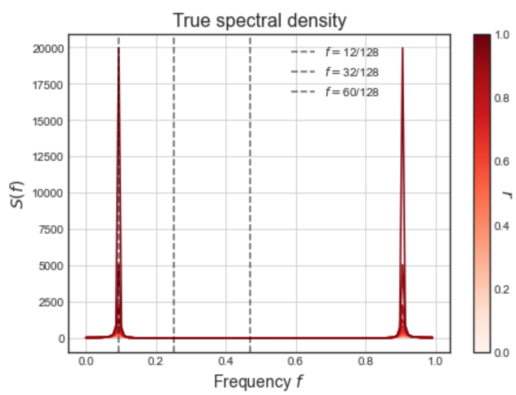


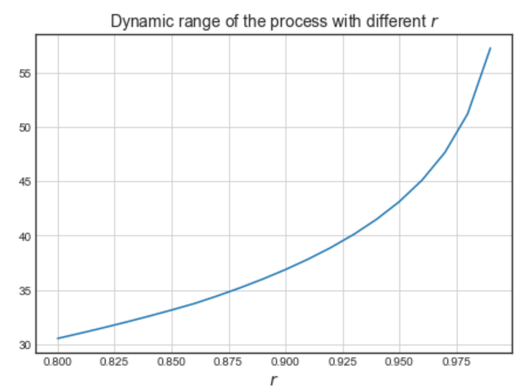
0.1.5 (e)

- From the two figures below, one can see that when r gets closer to 1, the process has larger dynamic range. As a result, for each estimator of the spectual density, there is more sidelobe leakage when r is larger, resulting in larger sample bias.
- At the oscillating frequency $f = 12/128$, the periodogram and the tapered estimators have similar sample bias.
- At the two frequencies away from the oscillating frequency, the tapered estimators have smaller bias than periodogram. In particular, more tapering (larger p) results in larger bias reduction.
- At $f = 32/128$, the 25% and 50% taperered estimators are able to avoid sidelobe leakage

almost completely. At $f = 60/128$ (even further away from the oscillating frequency), all four cosine tapered estimates have very low leakage for r that is not extremely close to 1.

[ ]:
```python
## True sdf
from matplotlib import cm

fss = np.arange(128) / 128
sdf = np.array([S_ARMA(fss, sigma2=1, phis=phi, thetas=thetas) for phi in phis])

plt.figure(figsize=(16, 5))
plt.subplot(121)
for k in range(20):
    plt.plot(fss, sdf[k], c=cm.Reds(k/20), lw=1.5)
sm = plt.cm.ScalarMappable(cmap=cm.Reds)

for f in fs:
    plt.axvline(x=f, alpha=0.6, c='k', linestyle='--',␣
↪label=r'$f=$'+f'{int(f*128)}/128')
cbar = plt.colorbar(sm)
cbar.set_label(r'$r$', rotation=270, fontsize=14)

plt.xlabel(r'Frequency $f$', fontsize=14)
plt.ylabel(r'$S(f)$', fontsize=14)
plt.title('True spectral density', fontsize=16)
plt.legend(loc=9, bbox_to_anchor=(0.7, 1), fontsize=10)
plt.grid()


## Dynamic range
plt.subplot(122)
dynamic_range = 10 * np.log10(sdf.max(-1) / sdf.min(-1))
plt.plot(rs, dynamic_range)
plt.title(r'Dynamic range of the process with different $r$', fontsize=14)
plt.xlabel(r'$r$', fontsize=14)
plt.grid()
plt.show()
```

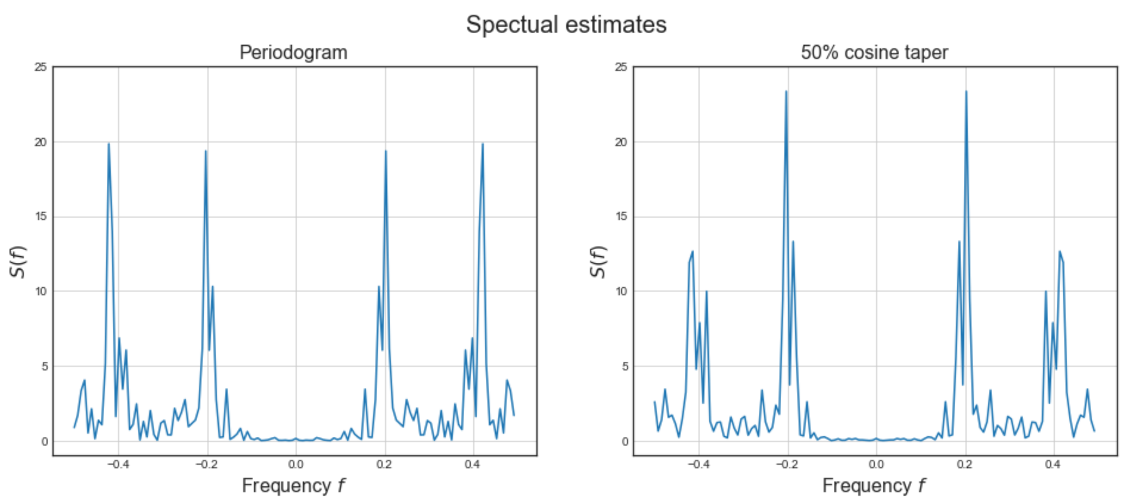




0.2 2

0.2.1 (a)

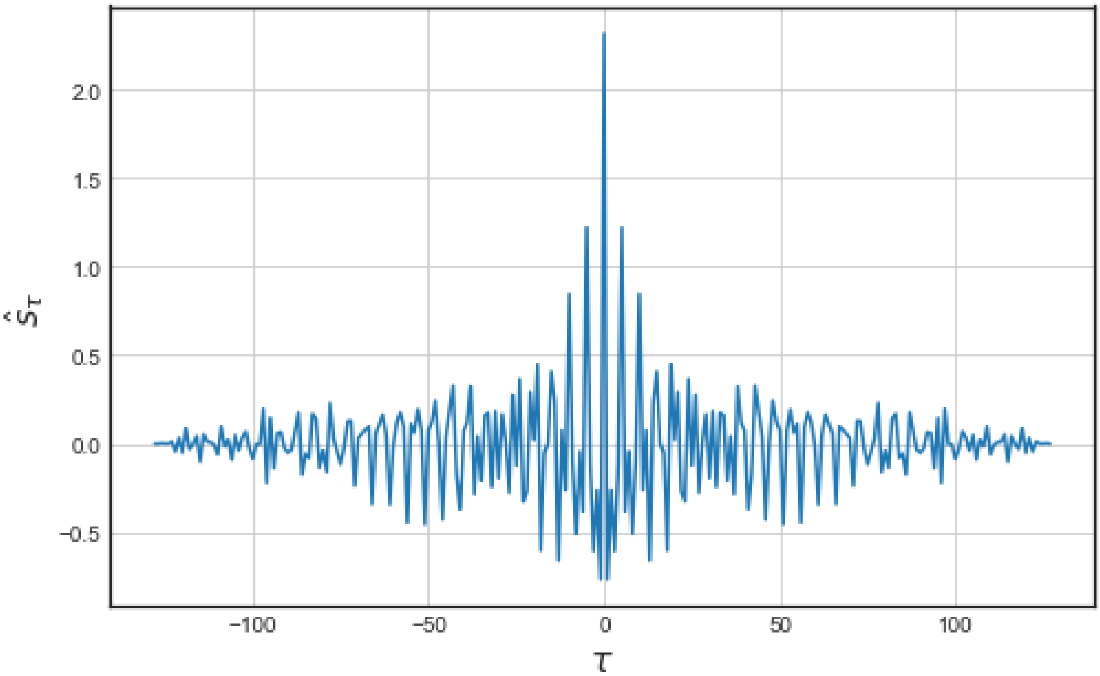
```
[ ]: X2 = np.genfromtxt('154.csv', delimiter=',')
N = X2.shape[-1]

per = np.fft.fftshift(periodogram(X2))
dse = np.fft.fftshift(direct(X2, 0.5))
titles = ['Periodogram', '50% cosine taper']

fig = plt.figure(figsize=(16, 6))
for i, est in enumerate([per, dse]):
    plt.subplot(1, 2, i+1)
    plt.plot(np.arange(-1/2, 1/2, 1/128), est)
    plt.xlabel(r'Frequency $f$', fontsize=16)
    plt.ylabel(r'$S(f)$', fontsize=16)
    plt.title(titles[i], fontsize=16)
    plt.ylim(-1, 25)
    plt.grid()
fig.suptitle('Spectual estimates', fontsize=20)
plt.show()
```



0.2.2 (b)

[ ]:
```
from scipy.linalg import toeplitz

N = 128
def autocov_seq_hat(X, symmetry=False):
    """Compute the sample autocovariance for a time series."""
    N = X.shape[-1]
    taus = np.arange(-N, N) if symmetry else np.arange(N+1)
    Ts = np.abs(taus).astype(int)
    return np.array([(X[:N-T]*X[T:]).sum() / N for T in Ts])

stau_hat = autocov_seq_hat(X2, True)
plt.figure(figsize=(8, 5))
plt.plot(range(-N, N), stau_hat)
plt.ylabel(r'$\hat{s}_{\tau}$', fontsize=16)
plt.xlabel(r'$\tau$', fontsize=16)
plt.grid()
plt.show()


### Yule-Walker (untapered)
def estimators_YW(X, p):
    stau_hat = autocov_seq_hat(X)
    Ghat = toeplitz(stau_hat[:p])
    gamma_hat = stau_hat[1:p+1]
    phip_hat = np.linalg.inv(Ghat).dot(gamma_hat)
    sigma2_hat = stau_hat[0] - phip_hat.dot(stau_hat[1:p+1])
    return phip_hat, sigma2_hat

### Yule-Walker with 50% cosine taper

def estimators_YW_tapered(X, p, taper=0.5):
    hX = tapered_series(X, taper)
    N = hX.shape[-1]
    stau_hat = autocov_seq_hat(hX) * N
    Ghat = toeplitz(stau_hat[:p])
    gamma_hat = stau_hat[1:p+1]
    phip_hat = np.linalg.inv(Ghat).dot(gamma_hat)
    sigma2_hat = stau_hat[0] - phip_hat.dot(stau_hat[1:p+1])
    return phip_hat, sigma2_hat

### Approximate MLE

def estimators_MLE(X, p):
    F = toeplitz(X[p-1:N-1], X[p::-1])
    XX = X[p:]
```

```
    phat = np.linalg.inv(F.T @ F) @ F.T @ XX
    A = XX - F.dot(phat)
    sigma2_hat = A.T @ A / (N-2*p)
    return phat, sigma2_hat
```



0.2.3 (c)

[ ]:
```
def reduced_AIC(X, p, method='YW'):
    fmap = {
        'YW': estimators_YW,
        'YW taper': estimators_YW_tapered,
        'MLE': estimators_MLE,
    }
    N = X.shape[-1]
    s2 = fmap[method](X, p)[-1]
    return 2*p + N * np.log(s2)
```

[ ]:
```
methods = ['YW', 'YW taper', 'MLE']
YW_data = np.array([reduced_AIC(X2, pp) for pp in range(1, 21)])
YW_taper_data = np.array([reduced_AIC(X2, pp, 'YW taper') for pp in range(1,␣
 ↪21)])
MLE_data = np.array([reduced_AIC(X2, pp, 'MLE') for pp in range(1, 21)])

df = pd.DataFrame(
```

```
    {'YW':YW_data, 'YW taper':YW_taper_data, 'MLE':MLE_data},
).rename_axis('p').rename(lambda x: x+1, axis='index')
df
```

```
[ ]:            YW    YW taper        MLE
p
1   94.783802  101.207951  88.484364
2   88.539991   91.802297  60.189262
3   58.736744   57.057308   4.743230
4    1.434667  -13.417359  10.099120
5    3.402804  -11.420202   4.460794
6    3.883213  -10.309071  27.430608
7    5.722645   -8.343541  15.235844
8    7.525434   -7.086085  28.619181
9    7.697199   -5.566404  17.343578
10   9.214986   -4.194696  26.839051
11  11.158646   -2.194762  42.094840
12  12.929766   -0.436913  32.737438
13  12.199077   -0.184707  28.229796
14  14.188579    1.791134  23.914521
15  15.620487    3.224196  33.819087
16  17.616934    5.189858  37.658731
17  19.614910    7.126054  34.288293
18  19.785123    7.764564  28.436826
19  20.869850    6.129727  36.388368
20  22.256291    8.124802  37.051904
```

0.2.4 (d)

```
[ ]: def best_p(ps, AICs):
         return ps[np.argmin(AICs, -1)]

     ps = np.arange(1, 21)
     best_ps = best_p(ps, [YW_data, YW_taper_data, MLE_data])
     for i in range(3):
         print(f'Best choice of p based on {methods[i]} method = {best_ps[i]}')

     YW_coeffs = estimators_YW(X2, best_ps[0])
     YW_taper_coeffs = estimators_YW_tapered(X2, best_ps[1])
     MLE_coeffs = estimators_MLE(X2, best_ps[2])
```

```
Best choice of p based on YW method = 4
Best choice of p based on YW taper method = 4
Best choice of p based on MLE method = 5
```

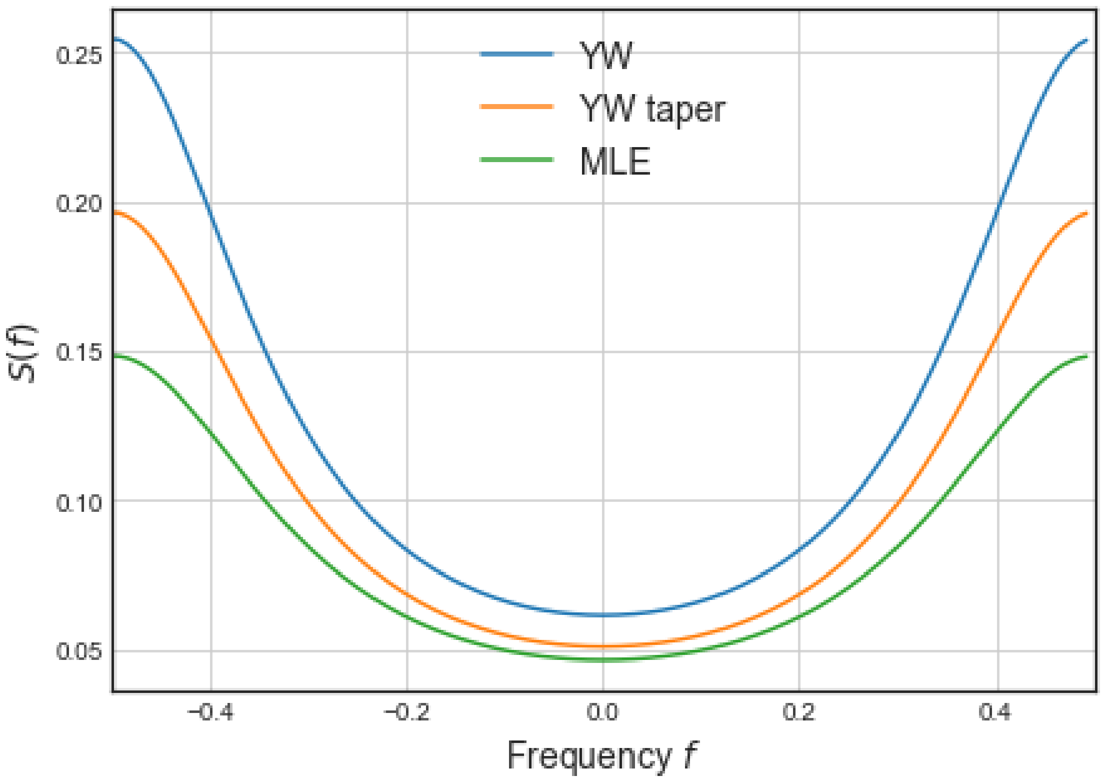
```
[ ]: print(f'Yule-Walker: Phi = {YW_coeffs[0]}, sigma^2 = {YW_coeffs[1]}')
     print(f'Yule-Walker with 50% taper: Phi = {YW_taper_coeffs[0]}, sigma^2 =␣
      ↪{YW_taper_coeffs[1]}')
```

```
print(f'Approximate MLE: Phi = {MLE_coeffs[0]}, sigma^2 = {MLE_coeffs[1]}')
```

```
Yule-Walker: Phi = [-0.81623113 -0.71385968 -0.79215113 -0.60892904], sigma^2 =
0.9500015475954233
Yule-Walker with 50% taper: Phi = [-0.82177909 -0.7716625  -0.82390842
-0.65751518], sigma^2 = 0.8459262307072268
Approximate MLE: Phi = [-0.86022111 -0.83303276 -0.91245236 -0.73306553
-0.10968231 -0.0919776 ], sigma^2 = 0.9576479573226758
```

0.2.5 (e)

[ ]:
```
plt.figure(figsize=(7, 5))
for i, data in enumerate([YW_coeffs, YW_taper_coeffs, MLE_coeffs]):
    plt.plot(
        np.arange(-1/2, 1/2, 1/128),
        S_ARMA(f=np.arange(-1/2, 1/2, 1/128), sigma2=data[-1], phis=data[:-1]),␣
 ↪label=methods[i]
    )
plt.xlabel(r'Frequency $f$',  fontsize=14)
plt.ylabel(r'$S(f)$',fontsize=14)
plt.legend(fontsize=14)
plt.xlim(-1/2, 1/2)
# plt.ylim(0, 0.3)
plt.grid()
plt.show()
```


YW
YW taper
MLE
0.25
0.20
0.15
0.10
0.05
S(f)
-0.4
-0.2
0.0
0.2
0.4
Frequency f

```
[ ]:
```