Lab_LinearModels1_v01_complete

October 10, 2024

0.1 EE512 – Applied Biomedical Signal Processing

1 Practical session – Linear Models I

1.0.1 Instructions

- This notebook provides all the questions of the practical session and the space to answer them. We recommend working directly here and then exporting the document as your report.
- Include any code used when addressing the questions, together with your answers.
- Please submit your report as a single PDF file.
- We recommend working in a group of 3–4 students; you must prepare one single report for the group (`name1_name2_name3_lab_LinearModelsI.pdf`), but every member needs to upload the same file individually.

[24]:
```
%matplotlib widget

# Imports
import os
import json
import numpy as np
import matplotlib.pyplot as plt

from statsmodels.tsa.ar_model import AutoReg, ar_select_order
from statsmodels.regression.linear_model import yule_walker

from scipy.signal import lfilter

# File paths
fbci = os.path.join(os.getcwd(), 'data', 'bci.json')
fafs = os.path.join(os.getcwd(), 'data', 'AF_sync.dat')
fspc = os.path.join(os.getcwd(), 'data', 'speech.dat')
fbld = os.path.join(os.getcwd(), 'data', 'blood.dat')
```

1.0.2 Experiment 1: classifying EEG signals

The file `/data/bci.mat` contains two data matrices, `left_hand` and `right_foot`, from a brain-computer interface (BCI) experiment. Each column in these matrices corresponds to a 2-second electroencephalography (EEG) recording (sampling frequency of 128 Hz) from the same electrode. The recordings in `left_hand` (respectively `right_foot`) were performed while the subject imagines a movement of the left hand (resp. right foot). The goal of the BCI experiment is to be able to "guess" what is being imagined based on the EEG signals alone.

We start by importing the signals (and removing their averages to better approximate our AR models):

[25]:
```python
with open(fbci, 'r') as f:
    jc = json.load(f)

eeg_lh = np.array(jc['left_hand'])
eeg_rf = np.array(jc['right_foot'])

eeg_lh -= np.mean(eeg_lh, axis=0, keepdims=True)
eeg_rf -= np.mean(eeg_rf, axis=0, keepdims=True)
```

AR model order A useful first step to look for structure in the signals is estimating the AR model order of each signal. We define a function `ar_order` to fit models of different orders, obtain the model noise variance, and apply a specific criterion:

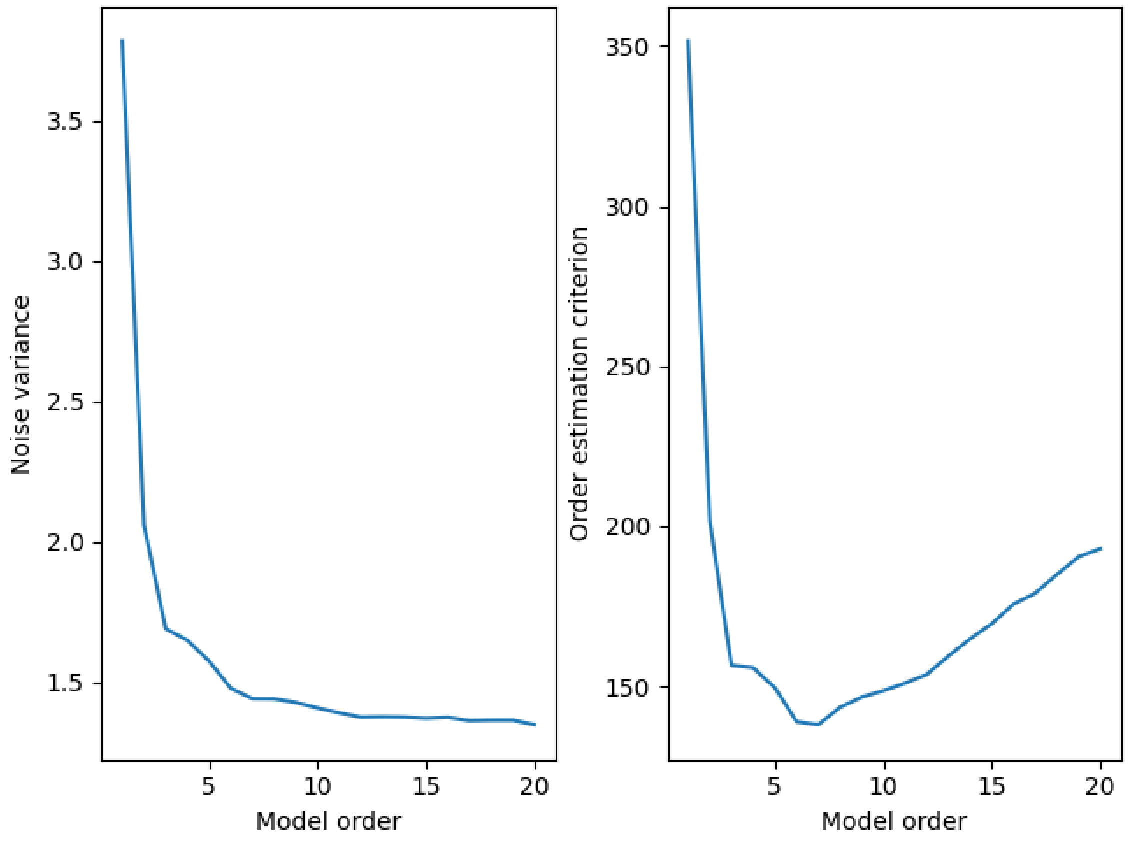
[26]:
```python
def ar_order(x, omax, Aff=0):
    """
    AR order estimation
    x: signal
    omax: maximum possible order
    Aff: 0 no graphic display; 1 display

    Returns:
    omdl: order estimated with MDL
    """

    nx = len(x)
    s = np.zeros((omax,))
    c = np.zeros((omax,))

    for k in range(omax):

        n = k+1

        ar_model = AutoReg(x, n, trend='n')
        ar_model_fit = ar_model.fit()
        sg2 = ar_model_fit.sigma2

        s[k] = sg2
        c[k] = nx * np.log(sg2) + (n+1) * np.log(nx)

    if Aff == 1:
        plt.figure()
        plt.plot(range(1, omax+1), mdl, 'o-')
        plt.title('Criterion')
        plt.show()

    return np.argmin(c)+1, s, c
```

Question 1.1. What is the name of the criterion being applied in the function `ar_order` implemented above? How would you change the code to apply the Akaike Information Criterion (AIC) instead?

Answer 1.1. It is the minimum description length criterion. To apply the Akaike Information Criterion, the definition of `c[k]` should be changed to `c[k] = nx * np.log(sg2) + 2*(n+1)`

Let's look at the order estimate for an example signal, in a reasonable range up 20:

[27]:
```
nmax = 20
_, s, c = ar_order(eeg_lh[:,1], nmax)

fig = plt.figure(constrained_layout=True)
n = np.arange(1, nmax+1)
plt.subplot(1,2,1)
plt.plot(n, s)
plt.xlabel('Model order')
plt.ylabel('Noise variance')
plt.subplot(1,2,2)
plt.plot(n, c)
plt.xlabel('Model order')
plt.ylabel('Order estimation criterion')
plt.show()
```



Question 1.2. Do the two curves obtained for this example signal behave as we would expect? What are their most important characteristics?

Answer 1.2. * The noise variances decreases with increasing model order. * The criterion decreases when the logarithm of noise variance decreases, but at the same time it increases with the logarithm of the model order. This can be seen in the plot as well as in the formula of the criterion: $\mathrm{MDL}(p) = N\ln(\sigma_p^2) + (p+1)\ln N$. As the noise variance decreases steeply for the first 7 model orders, the criterion plot has a visible minimum around that mark.

We can now estimate and print the optimal model order for every signal of each condition:

[28]:
```
nmax = 10

print("\nLeft hand :", end="")
for k in range(eeg_lh.shape[1]):
    n, _, _ = ar_order(eeg_lh[:,k], nmax)
    print("  {:02.0f}".format(n), end="")
print("")

print("Right foot:", end="")
for k in range(eeg_rf.shape[1]):
    n, _, _ = ar_order(eeg_rf[:,k], nmax)
    print("  {:02.0f}".format(n), end="")
print("\n")
```

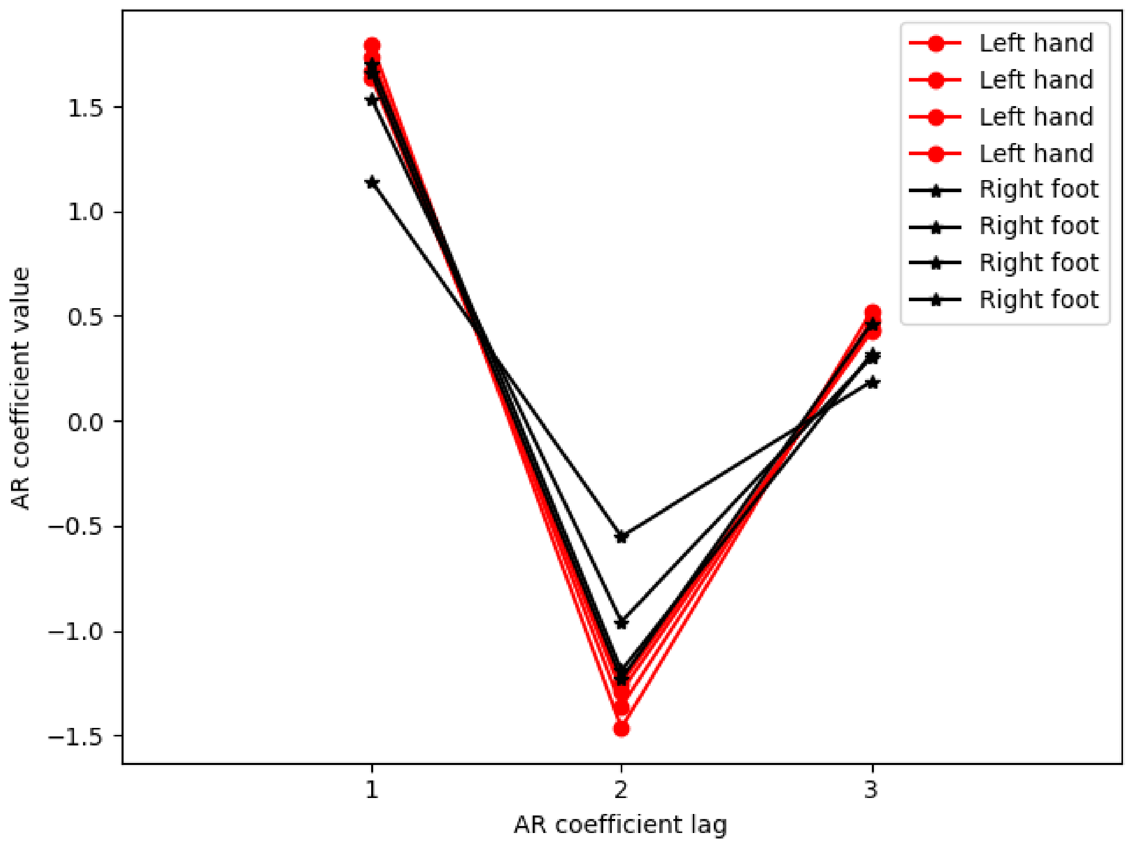
```
Left hand :  07  07  03  09
Right foot:  04  04  04  03
```

Question 1.3. Based on these AR order estimates, is there already a difference between the two categories overall? And if we wish to perform AR model estimation using a common choice of model order for all signals, which value should be chosen, and why?

Answer 1.3. The model order of 4 seem to be optimal for right foot signals, while left hand signals seem to require higher order models (order of at least 7). To perform AR model estimation for all signals, the smallest order should be used so that the resulting coefficients are significant and to avoid overfitting.

Movement prediction We now look at the resulting AR coefficients when choosing a model order of 3:

[29]:
```
n = 3

fig = plt.figure(constrained_layout=True)

for k in range(eeg_lh.shape[1]):
    ar_model = AutoReg(eeg_lh[:,k], n, trend='n')
    ar_model_fit = ar_model.fit()
    plt.plot(range(1,4), ar_model_fit.params, 'ro-', label='Left hand')

for k in range(eeg_rf.shape[1]):
    ar_model = AutoReg(eeg_rf[:,k], n, trend='n')
    ar_model_fit = ar_model.fit()
    plt.plot(range(1,4), ar_model_fit.params, 'k*-', label='Right foot')

plt.legend()
plt.xticks(ticks=(1,2,3))
plt.xlabel('AR coefficient lag')
plt.ylabel('AR coefficient value')
plt.xlim(0,4)

plt.show()
```



Question 1.4. On which coefficients is the separation between categories most promising?

Answer 1.4. On the coefficient with 2 samples lag; the values of this coefficient for the left hand signals clearly lie below the values that correspond to right foot signals.

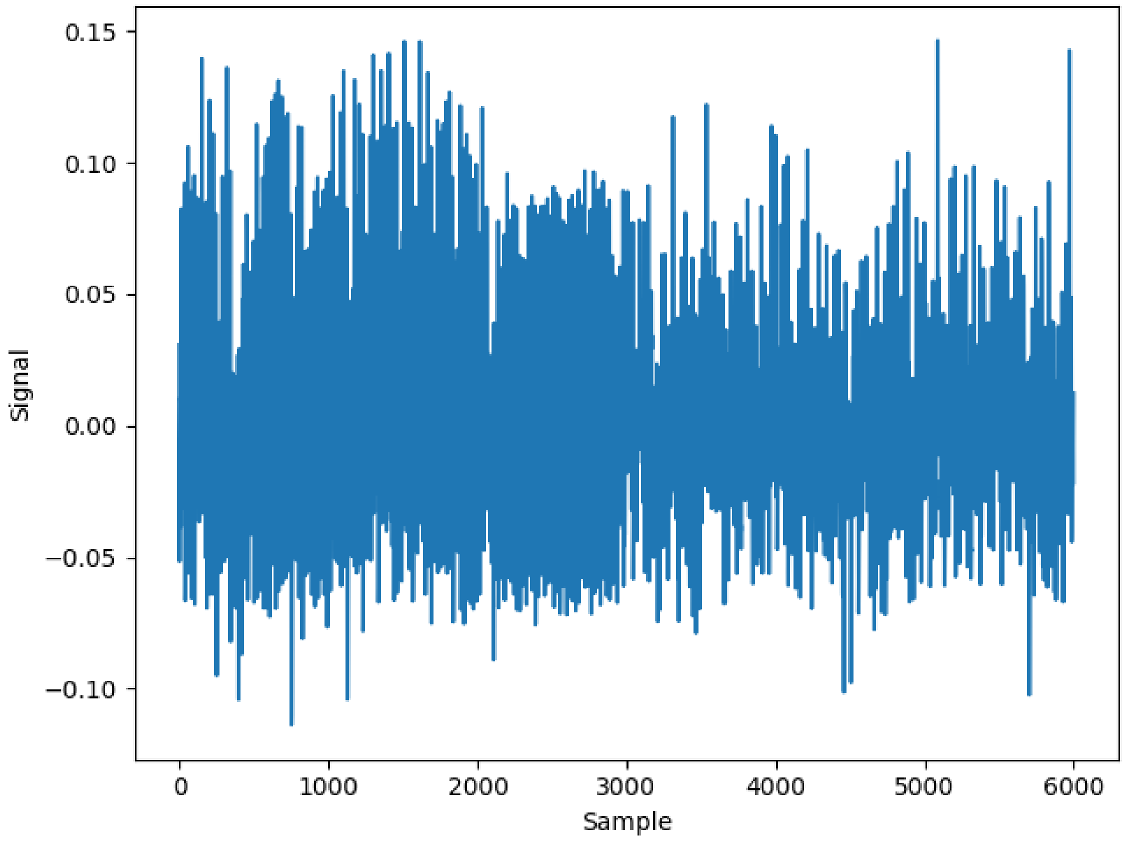
1.0.3 Experiment 2: AR model evolution over time

Real-life physiological signals can often vary substantially (and meaningfully) throughout a recording. It's usually good practice to have a look at the data before trying to apply models. Consider the signal in `AF_sync.dat` – a recording of ECG atrial activity during atrial fibrillation (sampling frequency of 50 Hz). We start by importing the signal:

[30]:
```
with open(fafs, 'r') as f:
    txt = f.readlines()
    af_sync = np.array([float(s[:-1]) for s in txt])

af_sync -= np.mean(af_sync)
```

Changes across time Let's plot the signal and consider its evolution over the course of the recording. At the start (until sample ~2000 approximately), the signal is moderately organized; then it becomes very organized until sample ~3000. This probably corresponds to a drastic reduction in the number of fibrillatory waves in the atrial tissue (flutter). In the last part of the recording, the fibrillation, and thus the signal, becomes very disorganized.

[31]:
```
fig = plt.figure(constrained_layout=True)
ax = plt.axes()
plt.plot(af_sync)
plt.xlabel('Sample')
plt.ylabel('Signal')
plt.show()
```

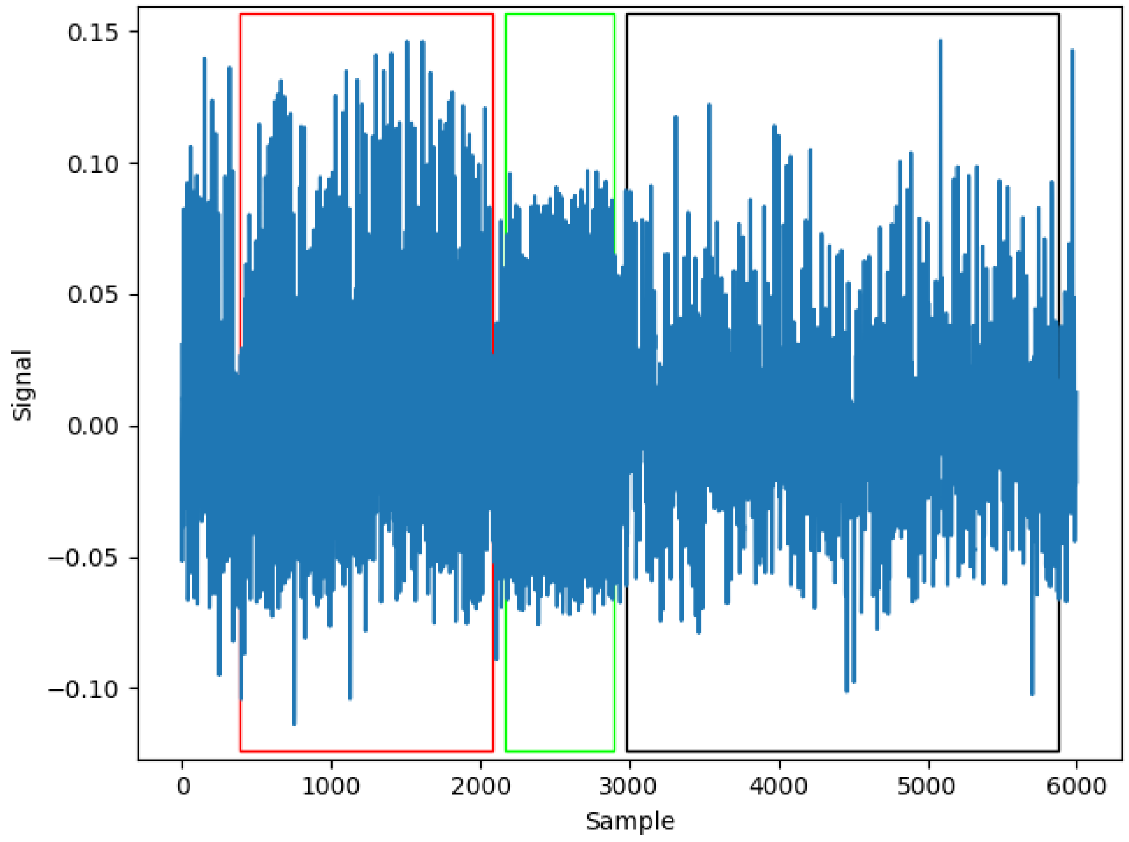


Question 2.1. The code below is intended to plot the signal again and mark the three periods described above, but it is not yet complete. Make the necessary modifications to show all three periods of interest.

Answer 2.1.

[32]:
```
fig = plt.figure(constrained_layout=True)
ax = plt.axes()
plt.plot(af_sync)
plt.xlabel('Sample')
plt.ylabel('Signal')
```

```
segments = [[383, 2078], [2170, 2890], [2973, 5881]]
eclr = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 0.0]]

ax.axvspan(segments[0][0], segments[0][1], ymin=0.01, ymax=0.99, ec=eclr[0],␣
 ↪fill=False)

ax.axvspan(segments[1][0], segments[1][1], ymin=0.01, ymax=0.99, ec=eclr[1],␣
 ↪fill=False)

ax.axvspan(segments[2][0], segments[2][1], ymin=0.01, ymax=0.99, ec=eclr[2],␣
 ↪fill=False)

plt.show()
```

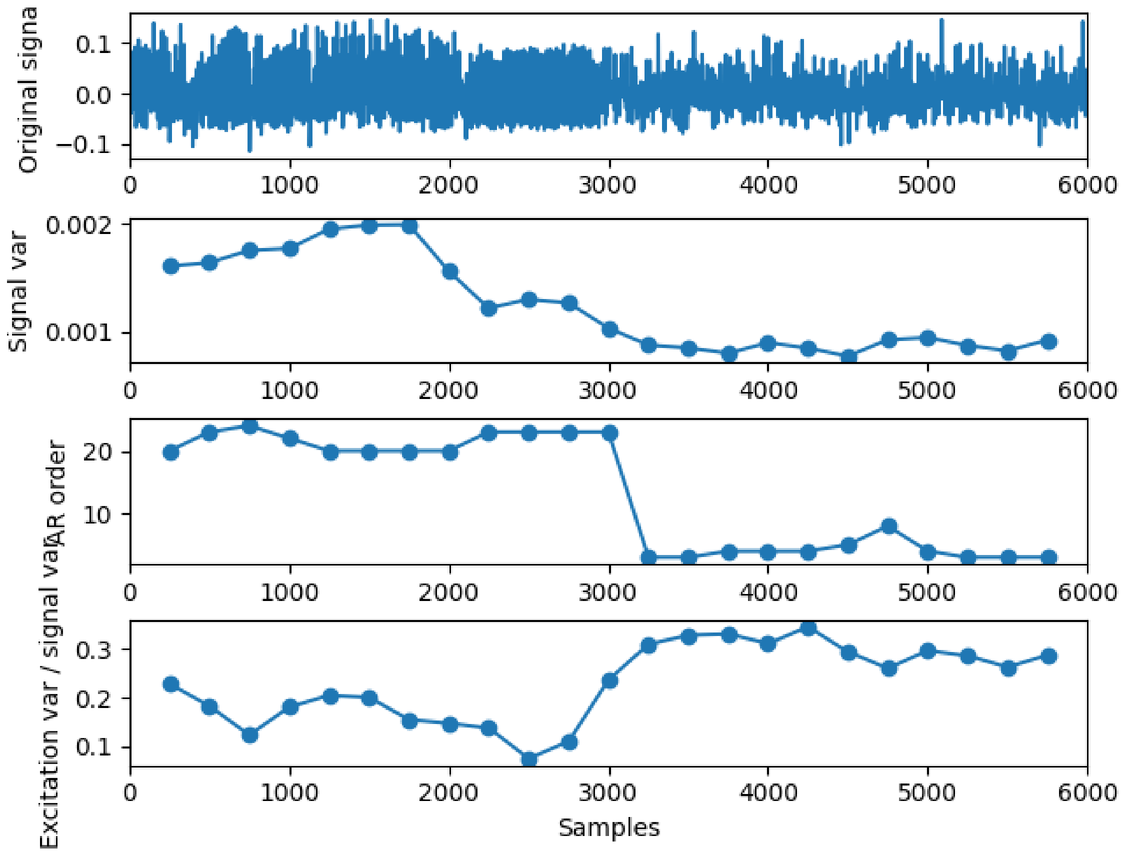


Adaptive modeling The clearly different states in `af_sync`, which vary across time, can be studied and modeled more quantitatively using a sliding-window approach. We thereby consider a segmentation of the signal into 500-sample windows with 50% overlap. For each segment, we can then estimate (i) the signal variance, (ii) optimal AR order, and (iii) the AR coefficients & excitation variance, as follows:

[33]:
```
nw = 500
nv = round(nw*0.50)
nt = len(af_sync)

kt = []
ki, kf = 0, nw

# Getting the border and middle indices for each segment
while True:
    kt.append([ki, round(0.5*(ki+kf)), kf])
    ki += nw - nv
    kf += nw - nv
    if kf > nt: break

nc = len(kt)
arv = np.zeros((nc,))
aro = np.zeros((nc,), dtype=int)
arr = np.zeros((nc,))
nmax = 40

# Modeling the signal for each segment
for kc in range(nc):

    ys = af_sync[kt[kc][0]:kt[kc][2]]

    arv[kc] = np.var(ys)
    aro[kc], _, _ = ar_order(ys, nmax)

    _, sigma = yule_walker(ys, order=int(aro[kc]), method="mle")
    arr[kc] = sigma**2 / arv[kc]
```

We can then plot together the time evolution of the raw signal, the signal variance, the AR order, and the ratio of excitation variance to signal variance.

[34]:
```
fig = plt.figure(constrained_layout=True)
t = np.array(kt)[:,1]

plt.subplot(4,1,1)
plt.plot(af_sync)
plt.ylabel('Original signal')
plt.xlim(0, len(af_sync))

plt.subplot(4,1,2)
plt.plot(t, arv, 'o-')
plt.ylabel('Signal var')
plt.xlim(0, len(af_sync))

plt.subplot(4,1,3)
plt.plot(t, aro, 'o-')
plt.ylabel('AR order')
plt.xlim(0, len(af_sync))

plt.subplot(4,1,4)
plt.plot(t, arr, 'o-')
plt.ylabel('Excitation var / signal var')
plt.xlim(0, len(af_sync))
plt.xlabel('Samples')

plt.show()
```

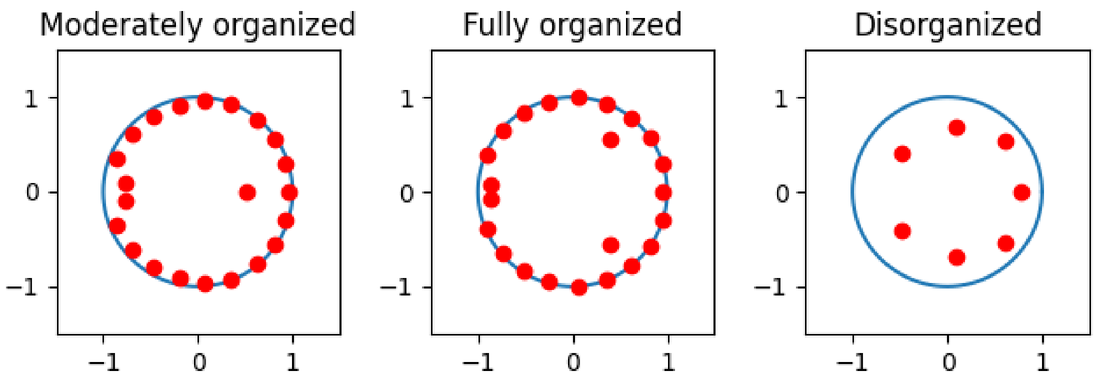


Question 2.2. Interpret the time evolution of the parameters plotted above, and how they relate to the organization of the signal in the three afore-mentioned stages.

Answer 2.2. As the signal gets more organized, it is better explained by an AR model with no excitation. Hence, the estimated AR order is higher, and the proportion of excitation variance compared to the signal variance is lower. At the ~2200 samples mark, we notice the increase in AR order and decrease in $\frac{\sigma_e^2}{\sigma_x^2}$, corresponding to the increase in organization of the signal. However, when the signal becomes disorganized (after ~3000 samples), the AR order drops low and the excitation variance goes up to describe the disorganized signal.

Signal stability and organization Finally, another way of evaluating how organized a signal is, is by looking at its equivalent filter transfer function *H(z)* and the positioning of the corresponding poles. We can focus specifically on the three states defined previously in `segments`, as follows:

[35]:
```python
roots = []

for seg in segments:

    ys = af_sync[seg[0]:seg[1]]

    aro, _, _ = ar_order(ys, nmax)
    rho, _ = yule_walker(ys, order=int(aro), method="mle")

    rho = np.concatenate((-rho[::-1], np.array([1])), axis=0)
    roots.append(1 / np.roots(rho))

fig = plt.figure(constrained_layout=True)
t = np.linspace(0, 2*np.pi, 1000)
titles = ['Moderately organized', 'Fully organized', 'Disorganized']

for k in range(3):
    plt.subplot(1,3,k+1)
    plt.plot(np.cos(t), np.sin(t))
    plt.plot(np.real(roots[k]), np.imag(roots[k]), 'ro')
    plt.axis('square')
    plt.xlim([-1.5, 1.5])
    plt.ylim([-1.5, 1.5])
    plt.title(titles[k])

plt.show()
```



And the proximity to the unit circle can be further summarized in terms of the average magnitude:

```
[36]: for kr in range(len(roots)):
          print(titles[kr] + ': {:.3f}'.format(np.mean(np.abs(roots[kr]))))
```

```
Moderately organized: 0.918
Fully organized: 0.951
Disorganized: 0.721
```

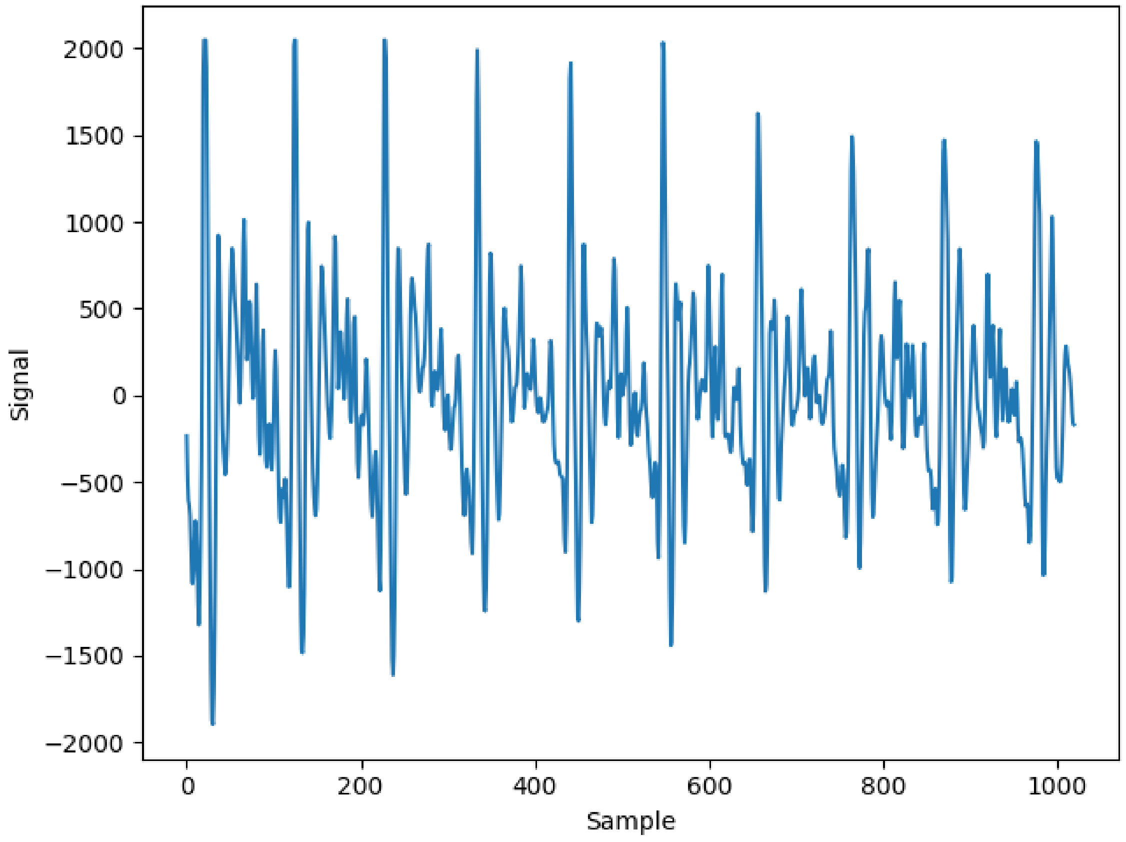
Question 2.3. Comment on how signal organization relates to pole location (proximity to the circle).

Answer 2.3. When the signal is organized, the poles are closer to the unit circle; this is visible both in the complex plot and in their mean magnitude which is close to 1. We can assume that a signal made of pure sinusoids would have its poles perfectly aligned on the unit circle. A disorganised signal has points that are further away from the unit circle.

1.0.4 Experiment 3: recovering the excitation (whitening filter)

The signal in `speech.dat` corresponds to the spoken sound /*a*/, sampled at 8 kHz. Used in language, we expect this signal to be clearly structured. We start by importing and plotting it:

[37]:
```
with open(fspc, 'r') as f:
    txt = f.readlines()
    speech = np.array([float(s[:-1]) for s in txt])

speech -= np.mean(speech)

fig = plt.figure(constrained_layout=True)
ax = plt.axes()
plt.plot(speech)
plt.xlabel('Sample')
plt.ylabel('Signal')
plt.show()
```



Using the functions `ar_order` and `yule_walker` introduced before, we can estimate the optimal model order for this signal, and then the corresponsing model parameters:

[38]:
```
aro1, _, _ = ar_order(speech, 40)
rho1, sgm1 = yule_walker(speech, order=int(aro1), method="mle")
```

The underlying excitation signal In the framework of AR modeling, the excitation signal driving the observed speech signal can be estimated using a filtering step as follows:

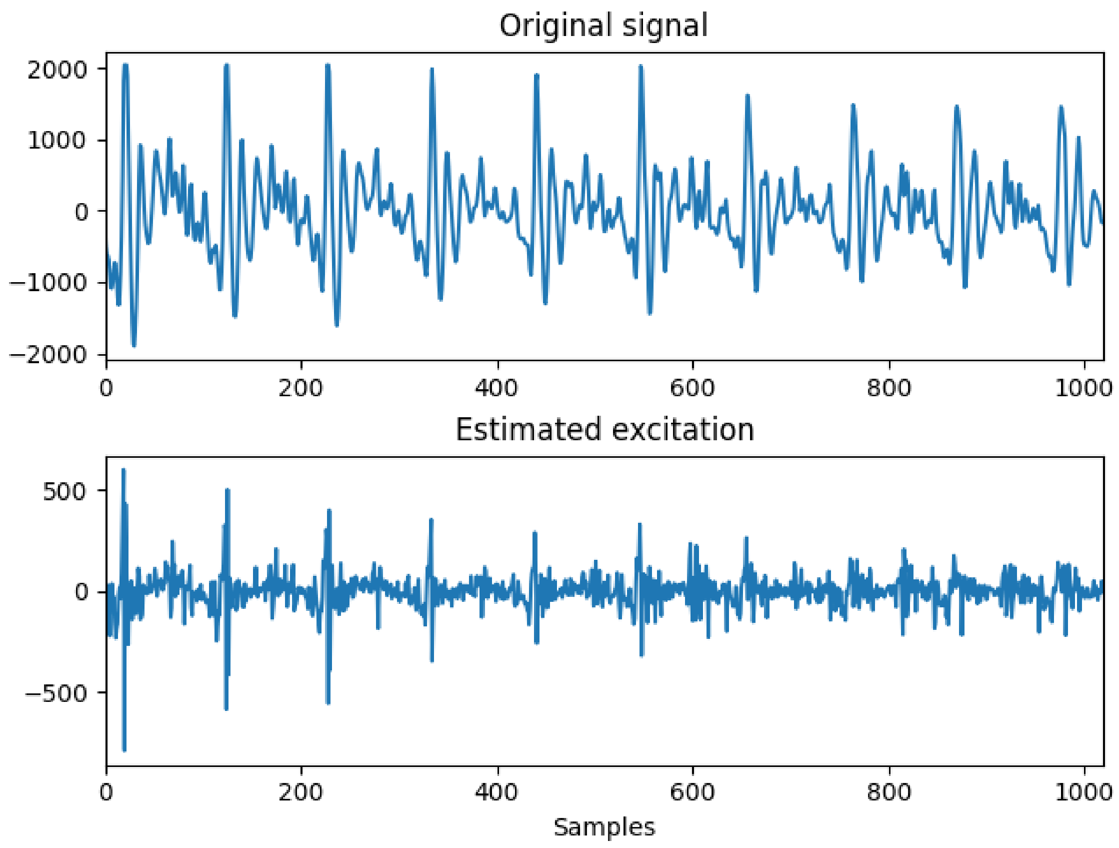
[39]:
```
exc1 = lfilter(np.concatenate(([1], -rho1), axis=0), [1], speech)
```

Question 3.1. Explain why the excitation can be estimated in this way.

Answer 3.1. The relationship between the speech signal and the excitation signal can be modeled in the z-domain as $X(z) = E(z)H(z)$, where $X(z)$ is the speech signal, $E(z)$ is the excitation signal and $H(z)$ is the filter that corresponds to the auto-regressive model. Therefore, one can write $E(z) = \frac{X(z)}{H(z)}$ and use the inverse filter $\frac{1}{H(z)}$ to estimate $E(z)$ given $X(z)$.

We can now visualize the estimated excitation signal:

[40]:
```
fig = plt.figure(constrained_layout=True)
plt.subplot(2,1,1)
plt.plot(speech)
plt.title('Original signal')
plt.xlim(0, len(speech))

plt.subplot(2,1,2)
plt.plot(exc1)
plt.title('Estimated excitation')
plt.xlim(0, len(speech))
plt.xlabel('Samples')
plt.show()
```



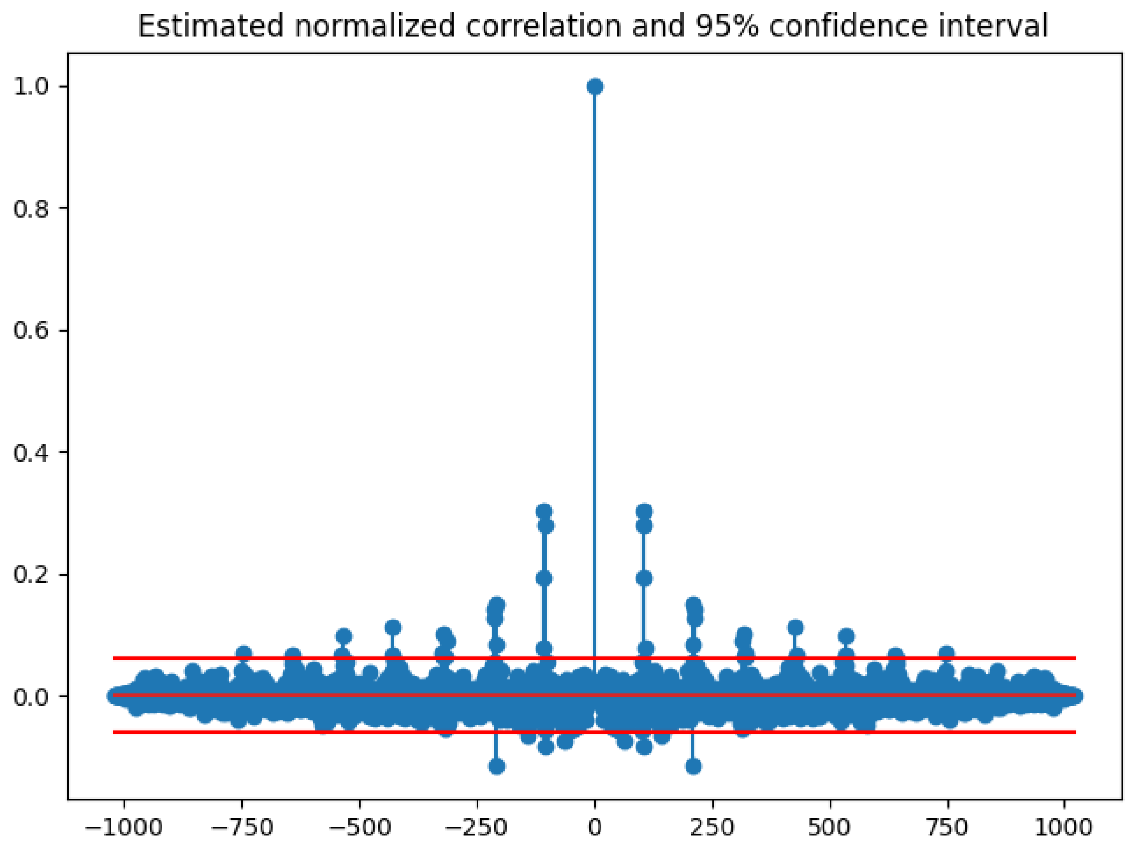
Question 3.2. Compare the excitation with the speech signal. Does the excitation look like white noise?

Answer 3.2. No. the estimated excitation clearly has a periodic pulse and does not appear stationary within the considered time window as the pulse decays in intensity over time, which is in contradiction with the assumption of the whiteness of the excitation process.

We can test more objectively whether the excitation is indeed similar to white noise by looking at its normalized autocorrelation, as follows:

```
[41]: def test_white(x, Aff=0):
          """
          Computation of the ratio of normalized autocorrelation estimates
          larger than a 5% threshold
          x: signal
          Aff: 0 no graphic display; 1 display
          """

          K = len(x)

          # Calculate the biased autocorrelation of the signal
          v = np.correlate(x, x, mode='full') / K
          # Note: K-1 is the index for zero lag

          thresh = 1.96 / np.sqrt(K)
          pc = np.sum(np.abs(v[K:] / v[K-1]) > thresh) / (K-1)

          if Aff == 1:
              fig = plt.figure(constrained_layout=True)
              ml = K-1 #min(30, K-1)
              lags = range(-ml, ml + 1)
              corr = v[K-1 - ml : K-1 + ml + 1] / v[K-1]
              plt.stem(lags, corr)
              plt.plot([-ml, ml], [+thresh, +thresh], 'r')
              plt.plot([-ml, ml], [-thresh, -thresh], 'r')
              plt.title('Estimated normalized correlation and 95% confidence␣
       ↪interval')
              plt.show()

          return pc

      print("Proportion above 5% threshold: {:f}".format(test_white(exc1, Aff=1)))
```

```
/var/folders/3q/bm92p78d1mqd90kqsld20cgr0000gn/T/ipykernel_12371/2118620198.py:1
9: RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained until
explicitly closed and may consume too much memory. (To control this warning, see
the rcParam `figure.max_open_warning`). Consider using
`matplotlib.pyplot.close()`.
  fig = plt.figure(constrained_layout=True)
```



```
Proportion above 5% threshold: 0.023553
```

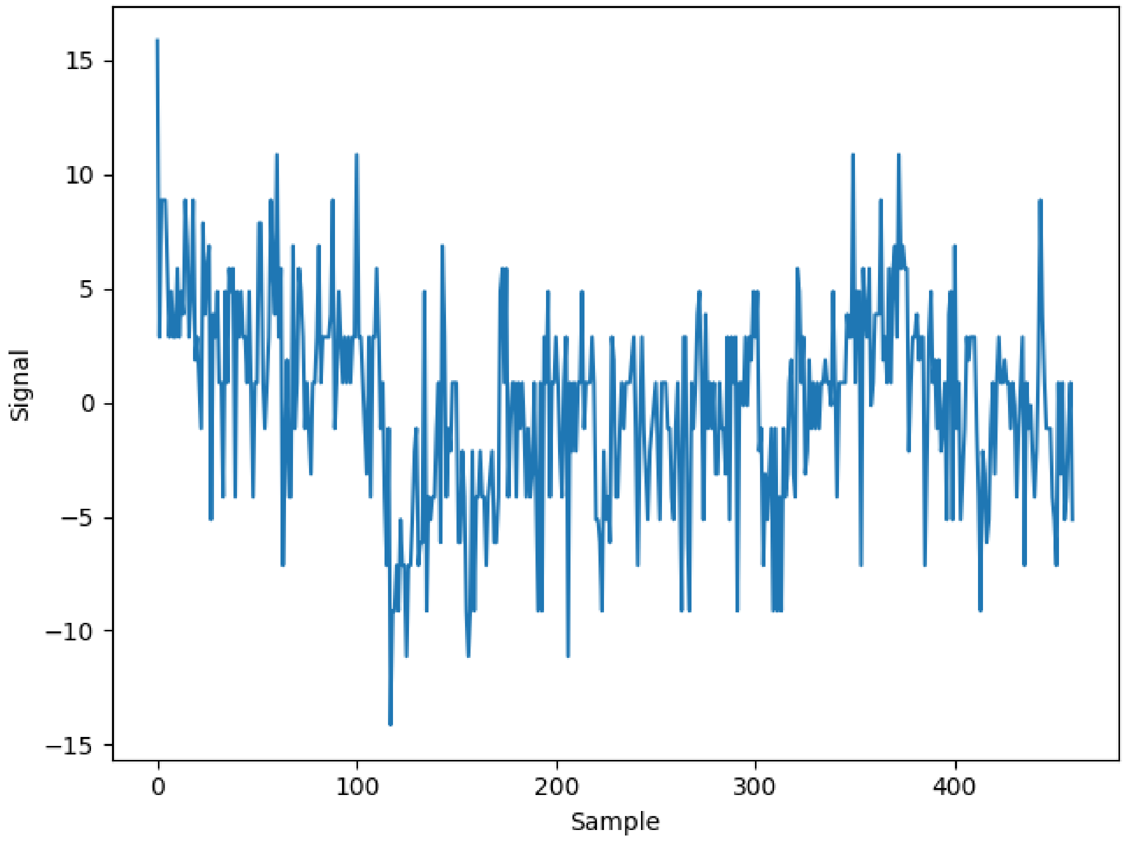
Question 3.3. What can we say based on this result?

Answer 3.3. Despite the visible periodic pulses, the autocorrelation is close to the one we would expect from white noise, as more than 97% of the autocorrelation are under the 5% threshold.

We now consider a different example: a timeseries of daily blood systolic pressure recorded from a patient, stored in `blood.dat`:

```
[42]: with open(fbld, 'r') as f:
          txt = f.readlines()
          txt = [s[:-1].split() for s in txt]
          blood = np.array([float(s[0]) for s in txt])

      blood -= np.mean(blood)

      fig = plt.figure(constrained_layout=True)
      ax = plt.axes()
```

```
plt.plot(blood)
plt.xlabel('Sample')
plt.ylabel('Signal')
plt.show()
```



Question 3.4. Repeat the full analysis done for the speech signal (i.e. estimating AR order, model parameters, excitation signal, whiteness test). How does this excitation signal compare to that of the speech example?

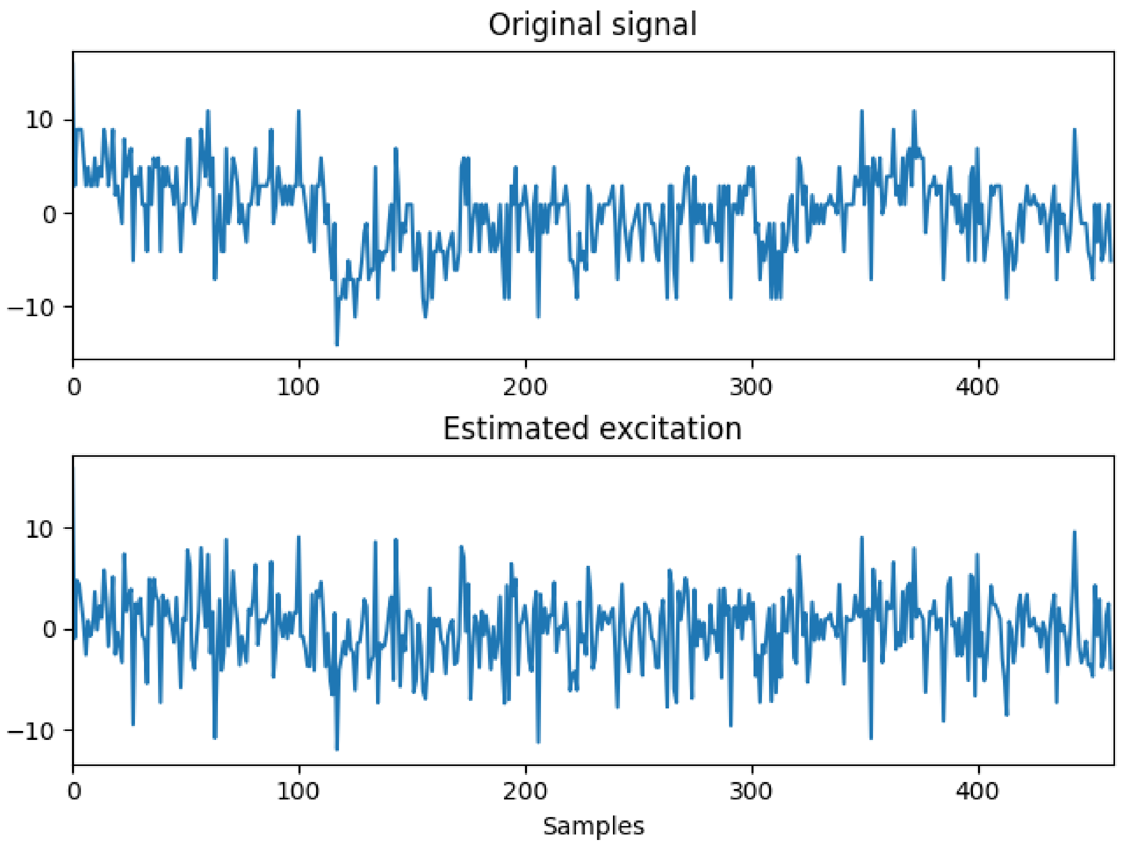
Answer 3.4. We find a model order estimate of 4, which is much lower than the order of 22 found for the speech example.

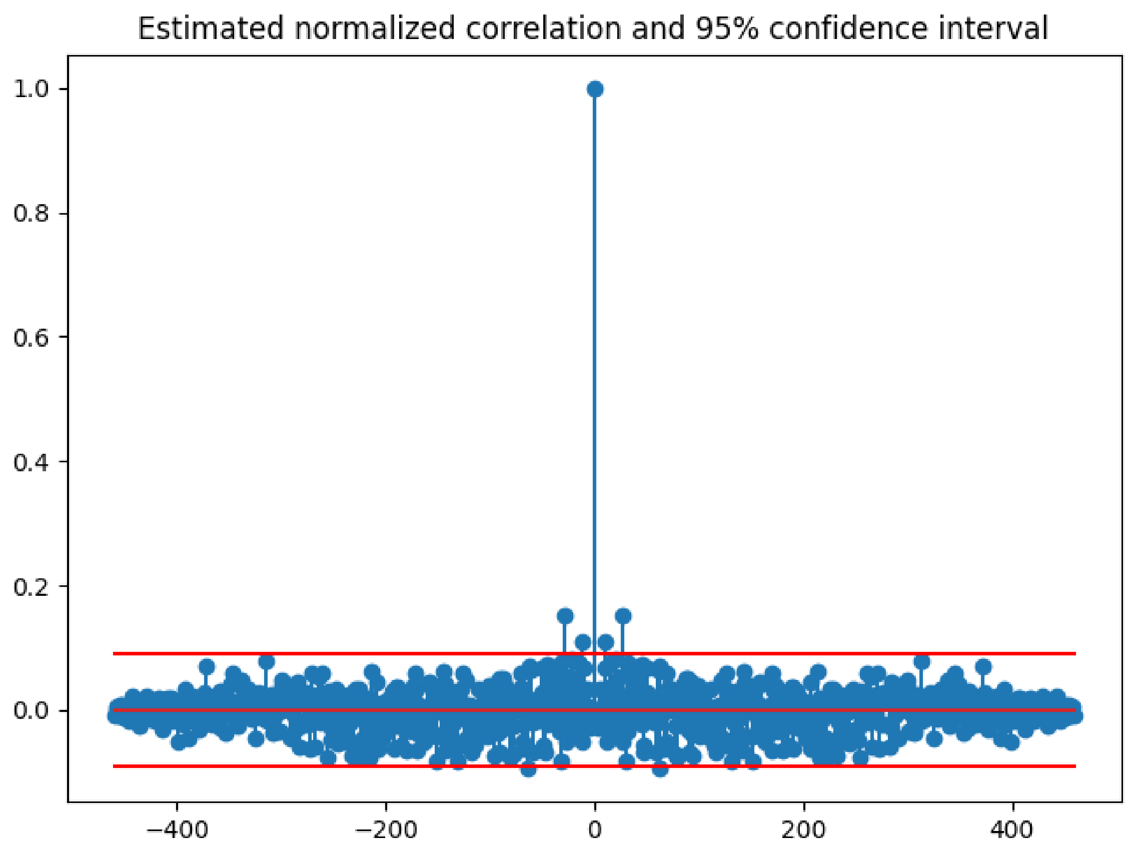
The resulting excitation signal looks more like white noise than the one of the speech signal, and this is confirmed by the fact that more than 99% of autocorrelation estimates are under the 5% threshold.

[43]:
```
aro1, _, _ = ar_order(blood, 40)
print(f"AR order estimate: {aro1}")
rho1, sgm1 = yule_walker(blood, order=int(aro1), method="mle")

exc1 = lfilter(np.concatenate(([1], -rho1), axis=0), [1], blood)
```

```python
fig = plt.figure(constrained_layout=True)
plt.subplot(2,1,1)
plt.plot(blood)
plt.title('Original signal')
plt.xlim(0, len(blood))

plt.subplot(2,1,2)
plt.plot(exc1)
plt.title('Estimated excitation')
plt.xlim(0, len(blood))
plt.xlabel('Samples')
plt.show()

print("Proportion above 5% threshold: {:f}".format(test_white(exc1, Aff=1)))
```

```
AR order estimate: 4
```





```
Proportion above 5% threshold: 0.006536
```

1.0.5 Have a good session, and don't hesitate to ask questions!