Difital Signal Processing
Homework 1

Anton Buguev BS19-RO-01
a.buguev@innopolis.university

10 April 2022

Task 1: Noise Reduction

1. We have 2 signals

$$y(t) = \sin(2\pi 50t) + \sin(2\pi 120t) \text{ and } y_\epsilon = y(t) + \epsilon \text{ for } t \in [0; 1]$$

where ϵ is random noise.

Since the maximum value of $\sin(\alpha) = 1$, which means that $\max(y(t)) = 2$, so let us make the range of random noise ϵ equal to $(-5; 5)$. This is implemented the following way:

```
import numpy as np
from matplotlib import pyplot as plt

n = 750 # number of subinervals
T = np.linspace(0, 1, n) # time array
y_clean = [] # values of y(t)
# calculate values of y(t) for each timestep
for t in T:
    y_clean.append(np.sin(2*np.pi*50*t) + np.sin(2*np.pi*120*t))
y_clean = np.array(y_clean) # convert list into numpy.array

eps = np.random.uniform(-5, 5, (n,)) # list of random errors
y_noise = y_clean + eps # list that contains values of y_eps(t)
```

Listing 1: Generate functions

Let us plot these 2 functions:

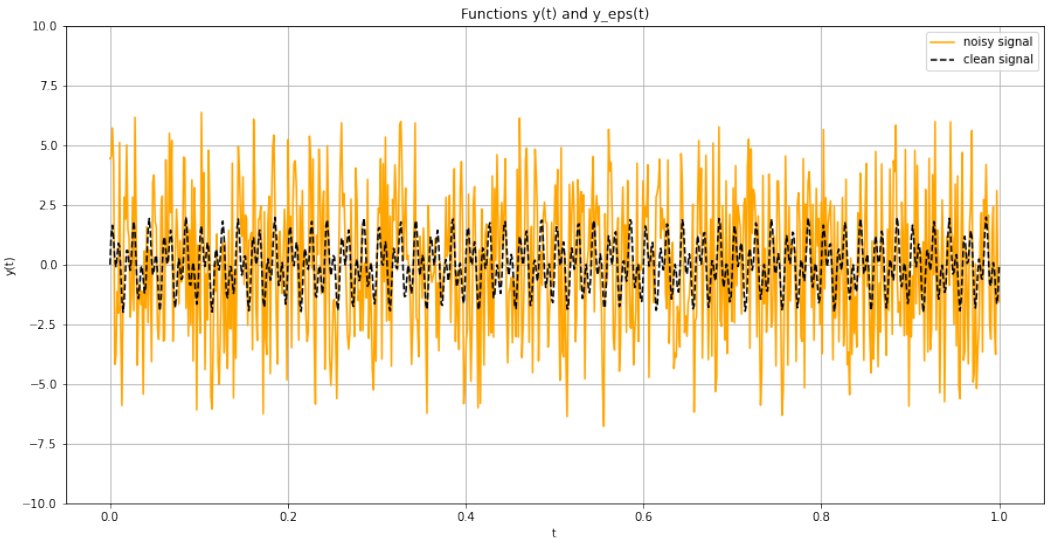

Figure 1: Original functions.

2. Now let us start procedure of removing noise from the function $y_\epsilon(t)$.

   First, let us calculate Fast Fourier Transform (FFT): $z = \mathcal{F}\{y_{eps}(t)\}(\omega)$. Python library *numpy* has function to calculate FFT so we do not have to implement it by ourselves.

```
z = np.fft.fft(y_noise) # FFT of y_eps(t)
```

Listing 2: FFT

3. After we calculated FFT let us calculate the Power Density Spectrum (PDS) of z by multiplying it element-wise by its conjugate, and dividing it by the number of points n, i.e.:

$$PDS = \frac{z\overline{z}}{n}$$

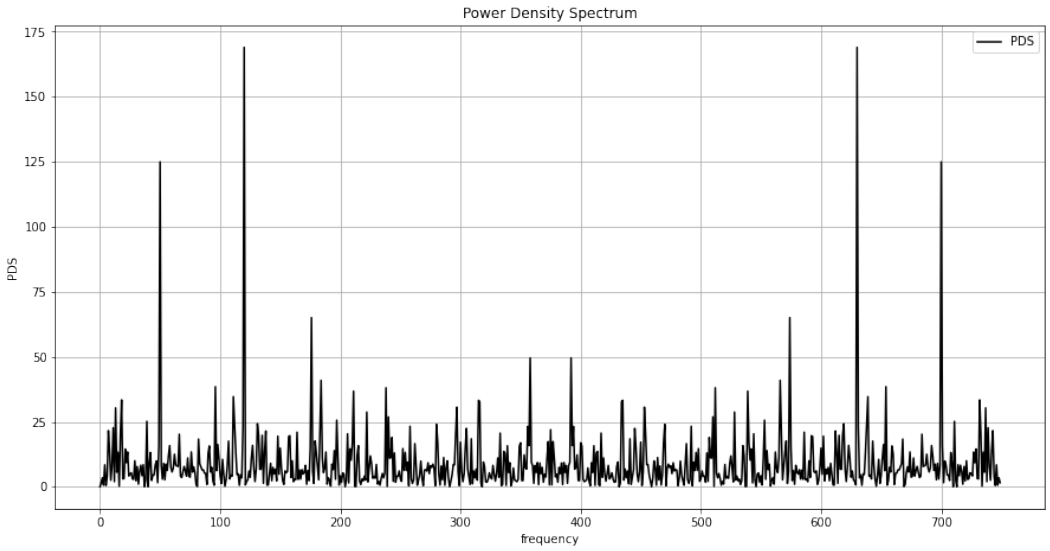

Figure 2: PDS.

4. The current data has very high peaks, so let us set threshold τ and keep indices of values that are lower than τ. Let us make $\tau = 100$ to remove the highest peaks.

```
tau = 100 # threshold
indices1 = [] # list of indices for the 1st task
# lists of PDS values, one will contain values which satisfy threshold,
# the second will contain elements which do not satisfy
# used for proper visualization
PDS_th, PDS_not = PDS.copy(), PDS.copy()
# Check every element of PDS
for i in range(len(PDS)):
    # if its value is smaller than our threshold,
    # then we keep its index and set to 0 corresponding element in list of not appropriate values
    # to visualize them later
    if PDS[i] < tau:
        indices1.append(i)
        PDS_not[i] = 0
    # otherwise, if it does not satisfy thershold,
    # we set to 0 correspondin element in list of appropriate values
    else:
        PDS_th[i] = 0
```

Listing 3: Signal filtering

After we check all elements we can visualize which what was removed.

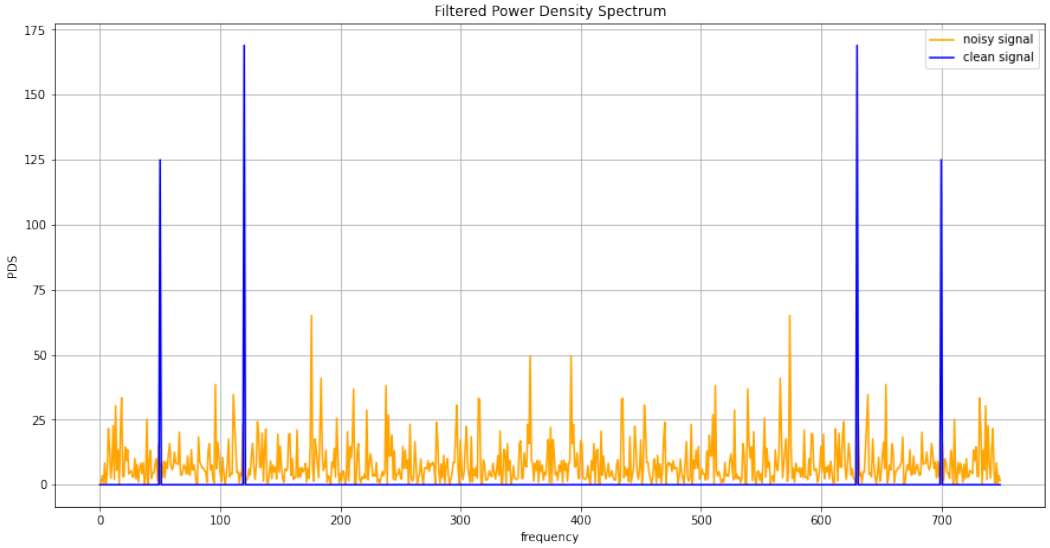

Figure 3: Signal Filtering.

5. After we saved necessary indices, let us update values of $z = \mathcal{F}\{y_{eps}(t)\}(\omega)$. If index of element of z was saved on previous step, therefore it has to be set to 0.

   Now let us calculate inverse FFT of z, i.e. $\mathcal{F}^{-1}(z)$, and see if all previous manipulations helped to remove noise:

```
z_upd = z.copy() # list of values of FFT which we will update
# if we saved an index of an element on previous step, then we have to set it to 0
z_upd[indices1] = 0

z_inv = np.fft.ifft(z_upd) # inverse FFT of updated values
```

Listing 4: Inverse FFT

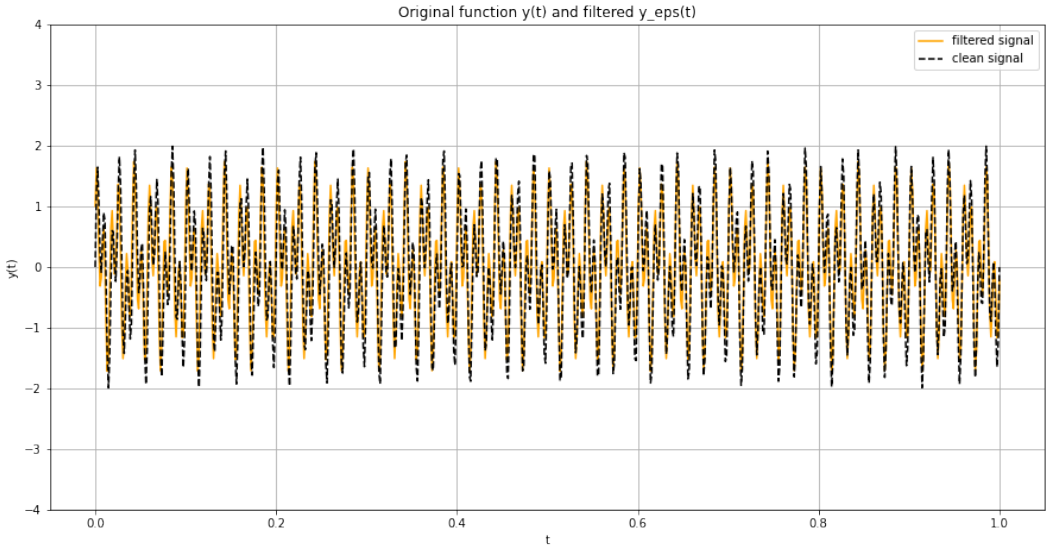

Figure 4: Noise reduction.

Conclusion: As we can see, noisy function $y_\epsilon(t)$ became much better, noise is removed and now it has the same plot as original $y(t)$ function, so we can conclude that we successfully removed noise despite the fact it was absolutely random.

Task 2: Image Compression

In this task we are going to compress the image.

Figure 5: Original image.

1. First, let us convert this image into grayscale format:

Figure 6: Grayscale image.

2. Now let us apply 2-dimensional FFT for this grayscale image to get the spectrum and shift zero-frequency to the center of the image. There is also special function in Python library *numpy* to calculate 2-dimensional FFT and shift zero-frequency to the center so again we do not have to implement this ourselves.

```
F2_sh = np.fft.fftshift(np.fft.fft2(A)) # shifted 2-dimensional FFT
```

Listing 5: 2D-FFT and zero-frequency shift

3. After that we need to calculate absolute value of each element of the shifted spectrum. If there are negative numbers, we will just get their absolute values. If there are complex numbers, then we will get their magnitude, i.e. if we have $z = a + bi$ we will get $|z| = \sqrt{a^2 + b^2}$:

4. Finally, we need to calculate natural logarithm of the previous computation adding 1 to each element and now we are ready to visualize the spectrum of the image.

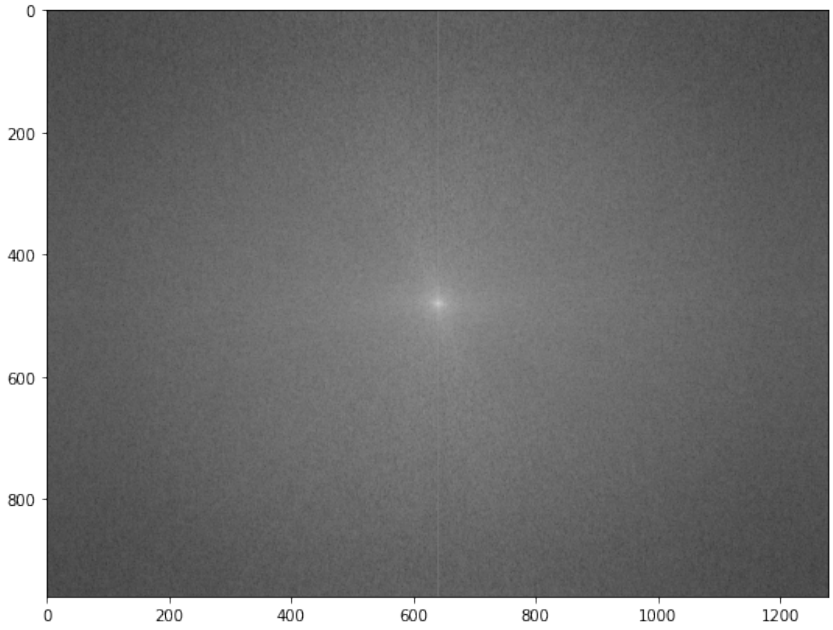

Figure 7: Image spectrum.

5. Now let us return to the original shifted matrix. For the further procedure we need to reshape it. The shifted matrix has sizes $w \times h$ or (w, h), where w and h are width and height of the image respectively. And we need to make new matrix with sizes $w \cdot h \times 1$ or $(w \cdot h, 1)$.

```
w, h = F2_sh.shape # sizes of image
F2_resh = np.reshape(F2_sh, (w * h, 1)) # Reshaped shifted matrix
```

Listing 6: Reshape matrix

6. After that, as before, we need to calculate absolute value of each element of the reshaped array.

```
Af = np.abs(F2_resh) # absolute values of reshaped matrix
```

Listing 7: Find absolute values

7. Then we need to sort result array.

```
a = np.sort(Af, axis=0) # sorted matrix
```

Listing 8: Sort array of absolute values

8. Also we need to set value for the threshold using which we will keep or remove certain elements. Firstly, let us set value for $\tau << 1$. It will be used to calculate index of element using formula $b = \lfloor (1 - \tau) \cdot w \cdot h \rfloor$.

```
tau = 0.03 # coefficient to calculate index
b = int(np.floor((1 - tau) * w * h)) # index of threshold
```

Listing 9: Find index

9. Now we can select element in the sorted array with index b and this element will be our threshold.

```
c = a[b][0] # threshold value
```

Listing 10: Find threshold

10. Now let us return to the unsorted array with absolute values with sizes $(w \cdot h, 1)$. We need to check every element of that array, if it is smaller than threshold, then we keep its index and set the corresponding element of matrix of image spectrum ($F2\_log$) to 0. This is what we get if we visualize updated image spectrum.

```
indices2 = [] # list of indices of appropriate elements for 2nd task
F2_upd = np.reshape(F2_log, (w*h, 1)) # list of logarithm values which we will update
for i in range(w*h):
    # if element is less than threshold, then we keep its index
    if Af[i][0] < c:
        F2_upd[i][0] = 0.0
        indices2.append(i)
```

Listing 11: Filter spectrum

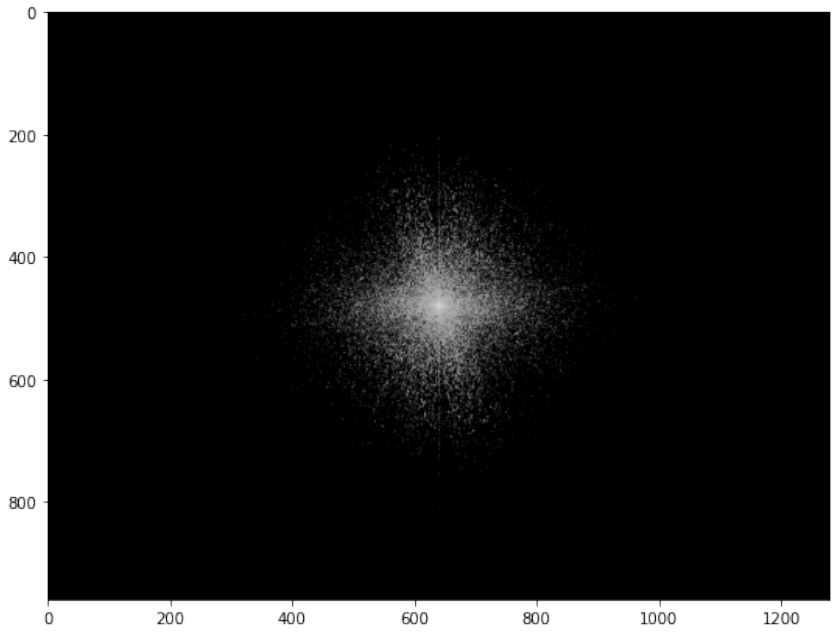

Figure 8: Filtered spectrum.

11. Finally, let us compress our initial image. To do that we need to take original shifted result of 2-dimensional FFT. Repeat the same procedure as before: find absolute value, add 1 to each element, find natural logarithm but only for those elements which indices we saved previously. After all that computation we need to do inverse shift, find inverse 2-dimensional FFT and visualize result image.

```
F2_final = np.reshape(F2_sh, (w*h, 1)) # reshape shifted FFT matrix to sizes (w*h, 1) for
    convenient work with indices
# for each element which index we saved we should find absolute value,
# add 1, calculate natural logarithm and update its value
F2_final[indices2] = np.log(np.abs(F2_final[indices2]) + 1)

F2_final = np.reshape(F2_final, (w, h)) # reshape updated matrix to sizes (w,h)
F2_final = np.fft.ifftshift(F2_final) # perform inverse shift
F2_final_inv = np.array(np.fft.ifft2(F2_final).real, dtype='uint8') # calculate inverse 2D-FFT
```

Listing 12: Reconstruct image

This is what we get as a result:

Figure 9: Original and reconstructed images.

Conclusion: As we can see, in the end we received an image that looks very similar to the original one. However, if we look closely we will notice that image actually is not that clear as it was before. It became blurred which means that we successfully compressed image.

And if we run code with different values of τ we will see different result: the smaller the value of τ, the more blurred image becomes:

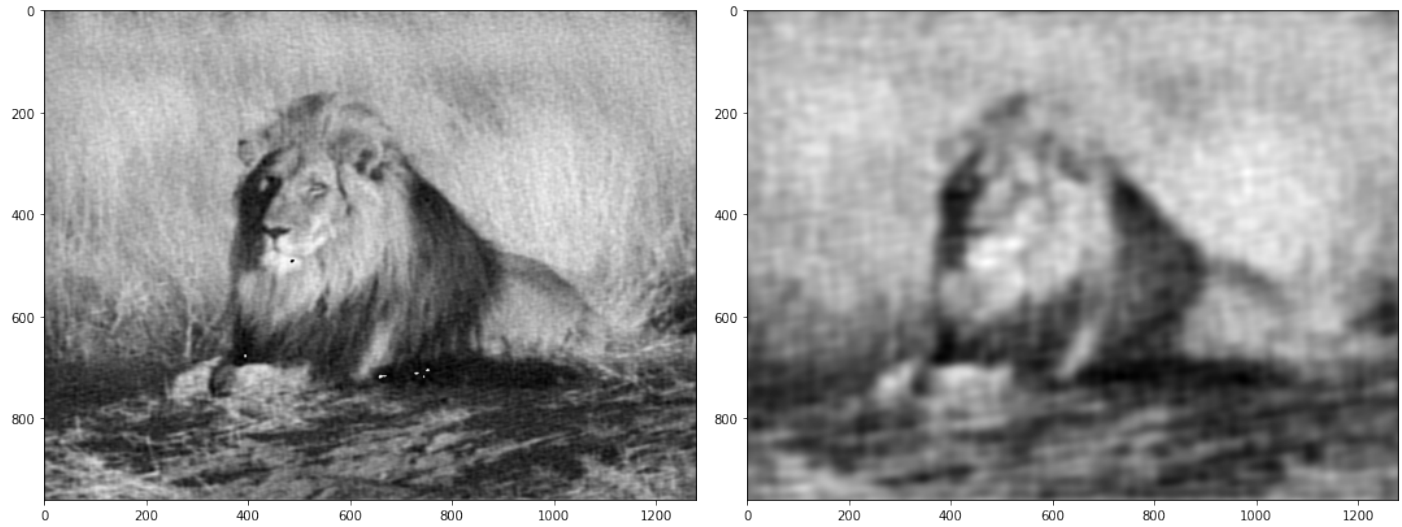

Figure 10: Reconstructed images when $\tau = 0.009$ and $\tau = 0.001$ respectively.