Colab Link: https://colab.research.google.com/drive/1AB-57JimmRNRDpXAvng-sFoiZuuBhmtn?usp=sharing

In [ ]:
```
!pip install statsmodels
```

```
Collecting statsmodels
  Downloading statsmodels-0.14.4-cp310-cp310-win_amd64.whl.metadata (9.5 kB)
Requirement already satisfied: numpy<3,>=1.22.3 in c:\users\nitro\appdata\local\programs\python\python310\lib\site-packages (from statsmodels) (1.23.5)
Requirement already satisfied: scipy!=1.9.2,>=1.8 in c:\users\nitro\appdata\local\programs\python\python310\lib\site-packages (from statsmodels) (1.14.1)
Requirement already satisfied: pandas!=2.1.0,>=1.4 in c:\users\nitro\appdata\local\programs\python\python310\lib\site-packages (from statsmodels) (2.2.3)
Collecting patsy>=0.5.6 (from statsmodels)
  Downloading patsy-1.0.1-py2.py3-none-any.whl.metadata (3.3 kB)
Requirement already satisfied: packaging>=21.3 in c:\users\nitro\appdata\local\programs\python\python310\lib\site-packages (from statsmodels) (23.2)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\nitro\appdata\roaming\python\python310\site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\users\nitro\appdata\local\programs\python\python310\lib\site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in c:\users\nitro\appdata\local\programs\python\python310\lib\site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2024.2)
Requirement already satisfied: six>=1.5 in c:\users\nitro\appdata\roaming\python\python310\site-packages (from python-dateutil>=2.8.2->pandas!=2.1.0,>=1.4->statsmodels) (1.16.0)
Downloading statsmodels-0.14.4-cp310-cp310-win_amd64.whl (9.8 MB)
   ---------------------------------------- 0.0/9.8 MB ? eta -:--:--
   --------- ------------------------------ 2.4/9.8 MB 11.2 MB/s eta 0:00:01
   ------------------------------ --------- 7.6/9.8 MB 18.8 MB/s eta 0:00:01
   ---------------------------------------- 9.8/9.8 MB 20.4 MB/s eta 0:00:00
Downloading patsy-1.0.1-py2.py3-none-any.whl (232 kB)
Installing collected packages: patsy, statsmodels
Successfully installed patsy-1.0.1 statsmodels-0.14.4
[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.signal import periodogram
from statsmodels.tsa.stattools import acf
```

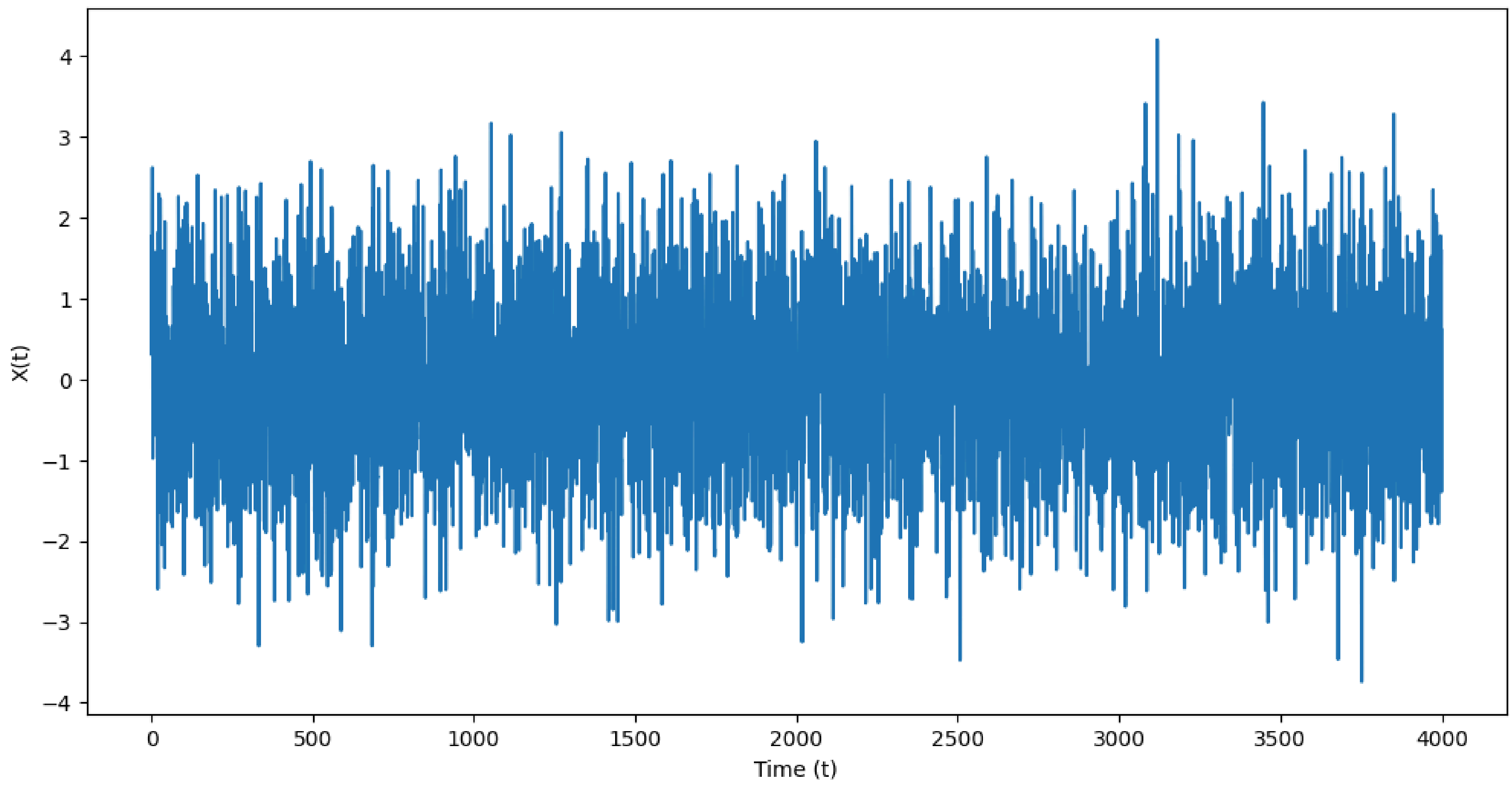
# 1

In [ ]:
```python
#1
#Initializing parameters
mean = 0
var = 1
T = 1
m = 5
delta_t = T/m

np.random.seed(0)

#Generate large set of Xk
X_k = np.random.normal(mean, var, 10000)

def k_t(t,T):
  return math.floor(t/T)
#We discard terms where k<0, when computing X(t)
def k_m(t,T,m):
  k = k_t(t,T)
  return max(0,k-m), max(0,k+m)

def generate_x_t(t, T, m, X_k):
  x_t = 0
  lower_bound, upper_bound = k_m(t,T,m)
  for k in range(lower_bound, upper_bound+1):
    x_t += X_k[k]*np.sinc((t-k*T)/T)
  return x_t
t_values = np.arange(0, 4000.1, 0.2)
X_t = []
for t in t_values:
  X_t.append(generate_x_t(t,T,m,X_k))
X_t[:5]
```

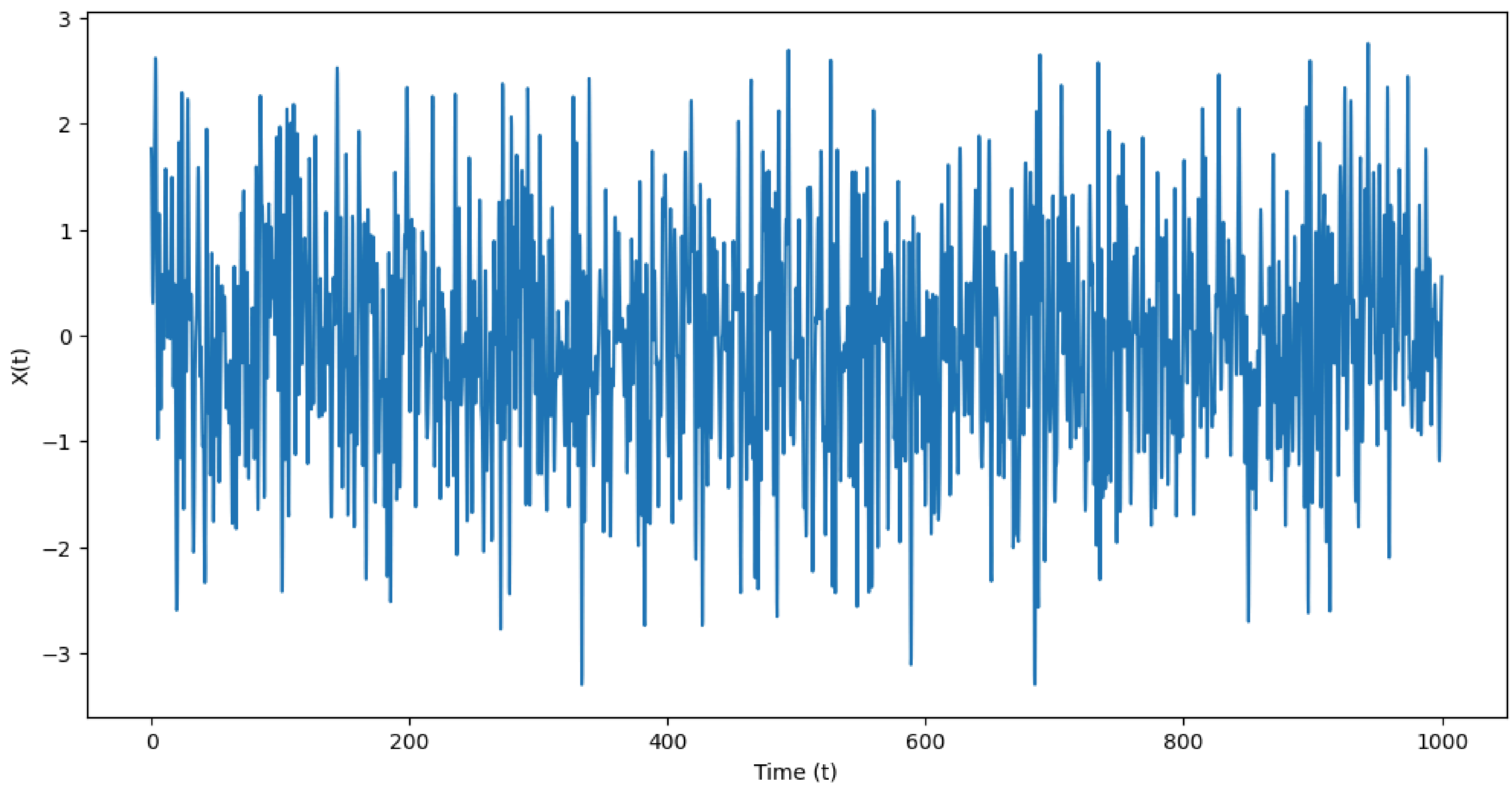
Out[ ]:
```
[1.764052345967664,
 1.6617978741316737,
 1.3913540371522115,
 1.0304034957366397,
 0.6721541439436208]
```

In [ ]:
```python
#Plotting X(t) for 3 different time scales
plt.figure(figsize=(12, 6))
```

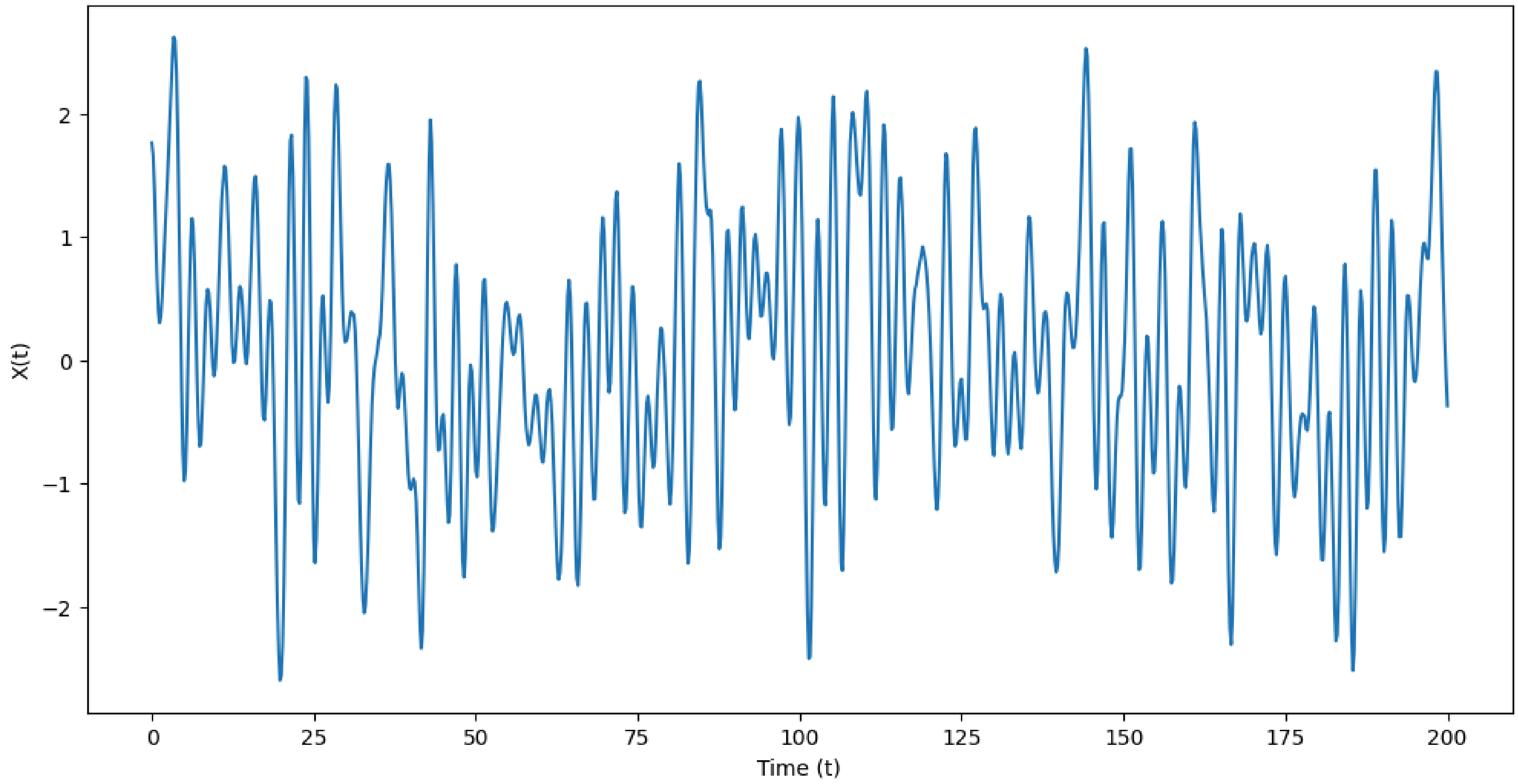
```
plt.plot(t_values, X_t)

plt.xlabel('Time (t)')
plt.ylabel('X(t)')
plt.show()
```




In [ ]:
```
plt.figure(figsize=(12, 6))
plt.plot(t_values[:5001], X_t[:5001])

plt.xlabel('Time (t)')
plt.ylabel('X(t)')
plt.show()
```



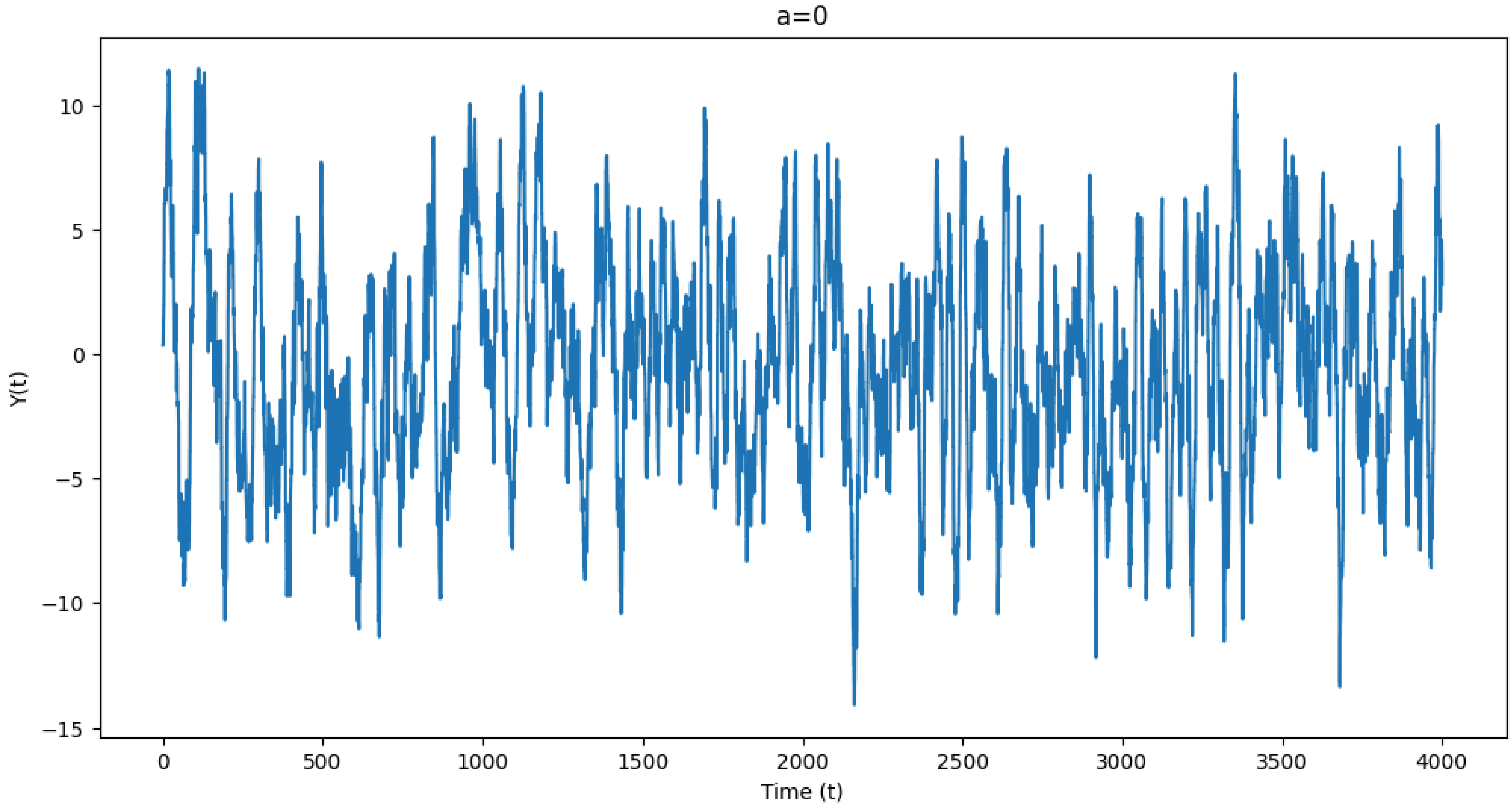

In [ ]:
```
plt.figure(figsize=(12, 6))
plt.plot(t_values[:1001], X_t[:1001])

plt.xlabel('Time (t)')
plt.ylabel('X(t)')
plt.show()
```



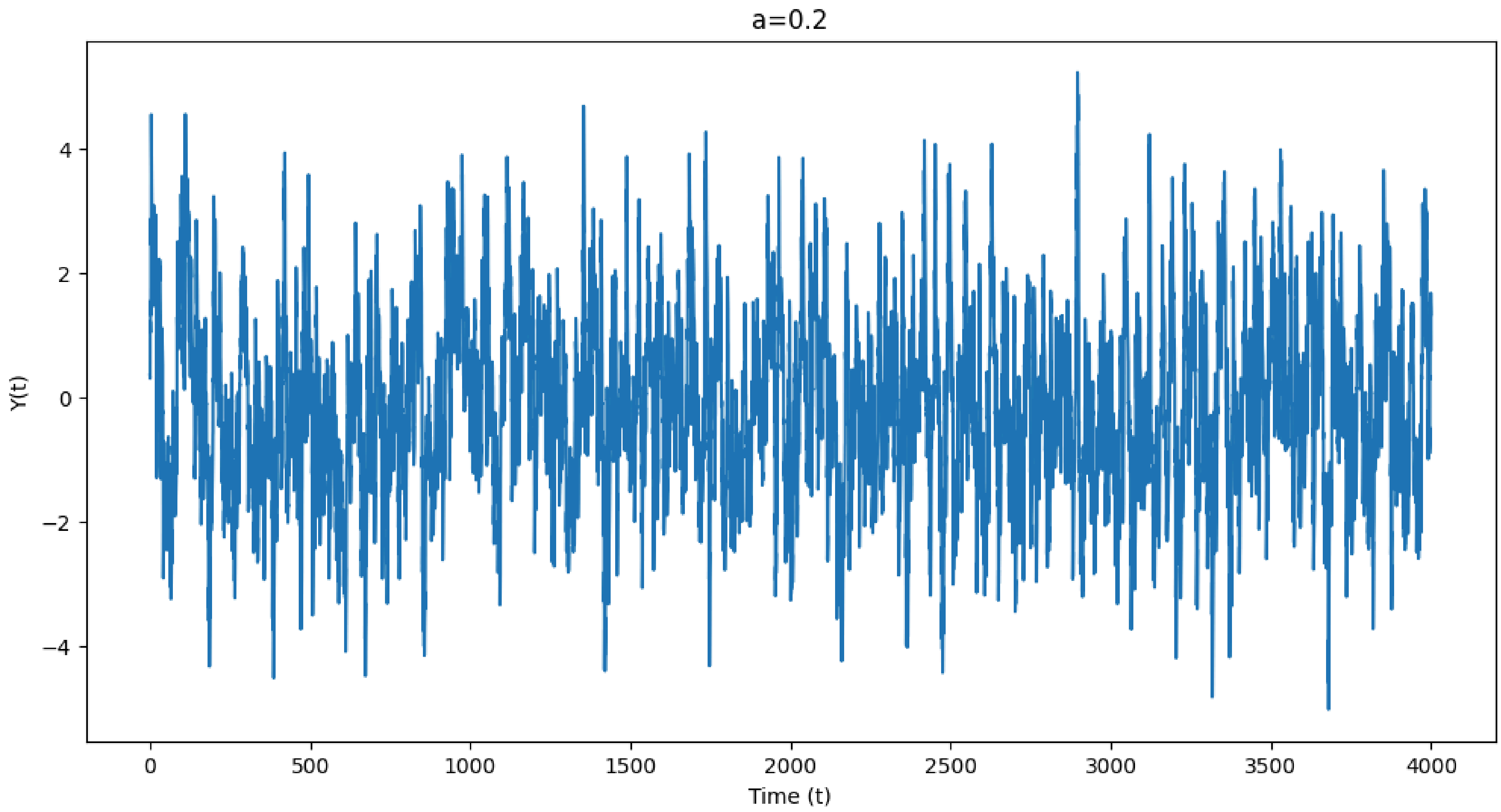

# 2

In [ ]:
```
#2

def generate_y_t(delta_t, m, X_k, t, a):
  #k = 0 -> 100
  k = np.arange(101)
  #Input to X(t)
  indices = np.full(101, math.floor(t))
  indices = indices - (k*delta_t)

  #X(t) = 0 for t < 0
  mask = indices >= 0

  #generate X(t) where t >= 0
  X = np.zeros(101)
  X[mask] = np.array([generate_x_t(idx, T, m, X_k) for idx in indices[mask]])

  exponent = (-a)*(t-math.floor(t)+(k*delta_t))
  y_t = X * np.exp(exponent)

  return delta_t * np.sum(y_t)
  # y_t = 0
  # for k in range(101):
  #   index = (math.floor(t) - k*delta_t)
  #   if index < 0:
  #     X = 0
  #   else:
  #     X = generate_x_t(index, T, m, X_k)
  #   exponent = (-a)*(t-math.floor(t)+(k*delta_t))
  #   y_t += X*math.exp(exponent)
  # return delta_t * y_t

def generate_Y(t_values, delta_t, m, X_k, a):
  return np.array([generate_y_t(delta_t, m, X_k, t, a) for t in t_values])
  # Y_t = []
  # for t in t_values:
  #   Y_t.append(generate_y_t(delta_t, m, X_k, t, a))
  # return Y_t
```

In [ ]:
```
#Case 1: a = 0
Y_t = generate_Y(t_values, delta_t, m, X_k, 0)
plt.figure(figsize=(12, 6))
plt.plot(t_values, Y_t)

plt.title("a=0")
plt.xlabel('Time (t)')
plt.ylabel('Y(t)')
plt.show()
```




In [ ]:
```
#Case 2: a = 0.2

Y_t_2 = generate_Y(t_values, delta_t, m, X_k, 0.2)
plt.figure(figsize=(12, 6))
plt.plot(t_values, Y_t_2)

plt.title("a=0.2")
plt.xlabel('Time (t)')
plt.ylabel('Y(t)')
plt.show()
```




# 3

For LTI systems, $S_y(f) = |H(f)|^2 * S_x(f)$, where $S_y(f)$ is the output PSD, $S_x(f)$ is the input PSD, and H(f) is the Fourier transform of h(t). To find $S_x(f)$, we look at the properties of bandlimited WGN process where the net power of X(t) is samples $X_k \sim N(0, N_0B)$ and B is the bandwidth. As $N_0 = 2$ and B = 1/2, $S_x(f) = N_0/2 = 1$ for $|f| < 1/2$. Note the formula of H(f) in the picture below and we can compute $|H(f)|^2$ by multiplying its conjugate.

$$H(f) = \frac{1-e^{-20(a+j2\pi f)}}{a+j2\pi f}$$

$$|H(f)|^2 = \frac{1 + e^{-20a}(e^2 - 2\cos(40\pi f))}{a^2 + 4\pi^2 f^2}$$

We substitute a with 0, 0.2 in the above equation and obtain

$$|H(f)|^2 = \frac{1 + (e^2 - 2\cos(40\pi f))}{4\pi^2 f^2}, a = 0$$

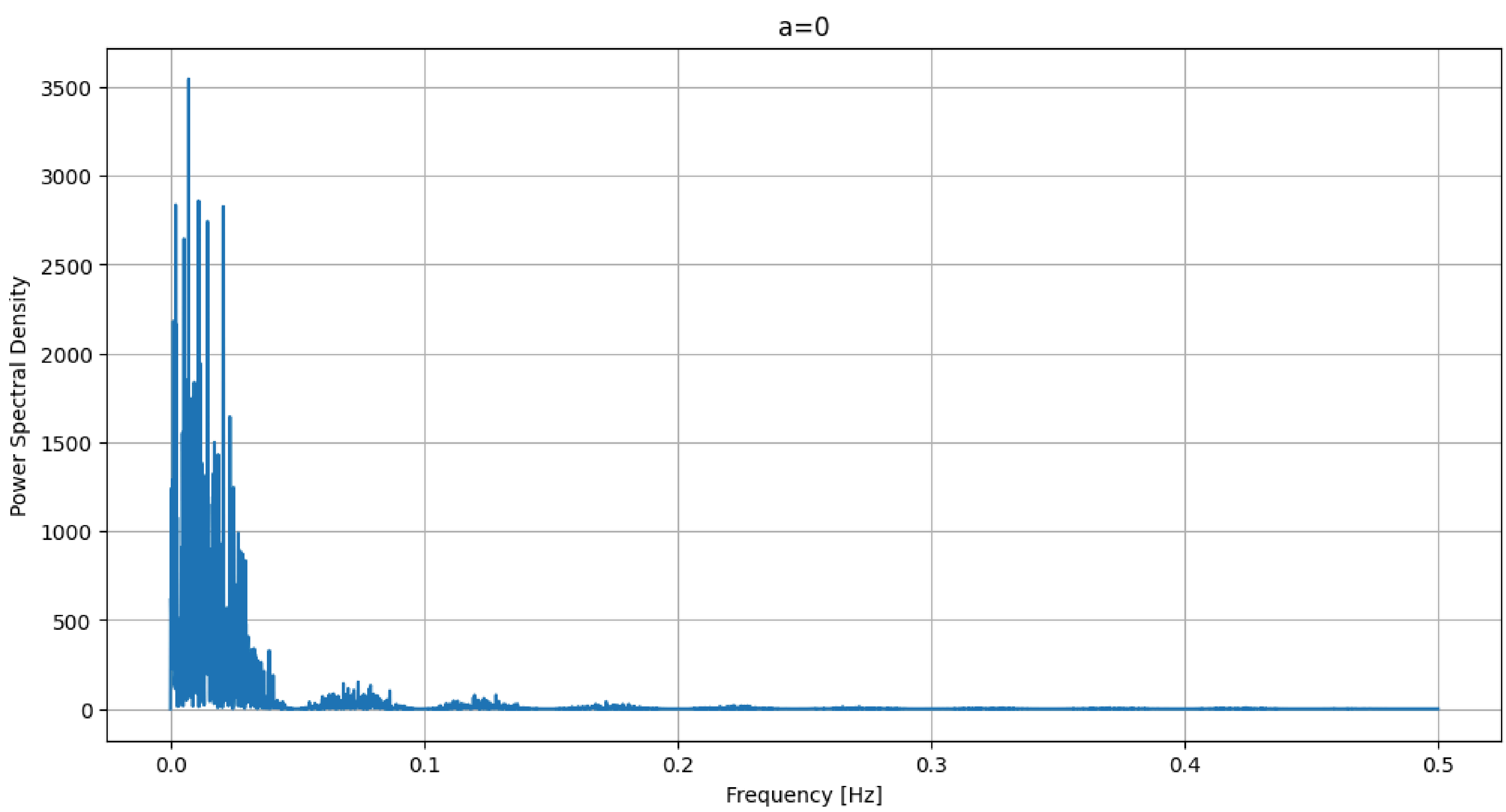
$$|H(f)|^2 = \frac{1 + e^{-4}(e^2 - 2\cos(40\pi f))}{0.04 + 4\pi^2 f^2}, a = 0.2$$

Therefore, $S_y(f) = |H(f)|^2 * 1$, for $|f| < 1/2$ and 0 otherwise. For a = 0, note that $S_y(f)$ can be undefined around f = 0.

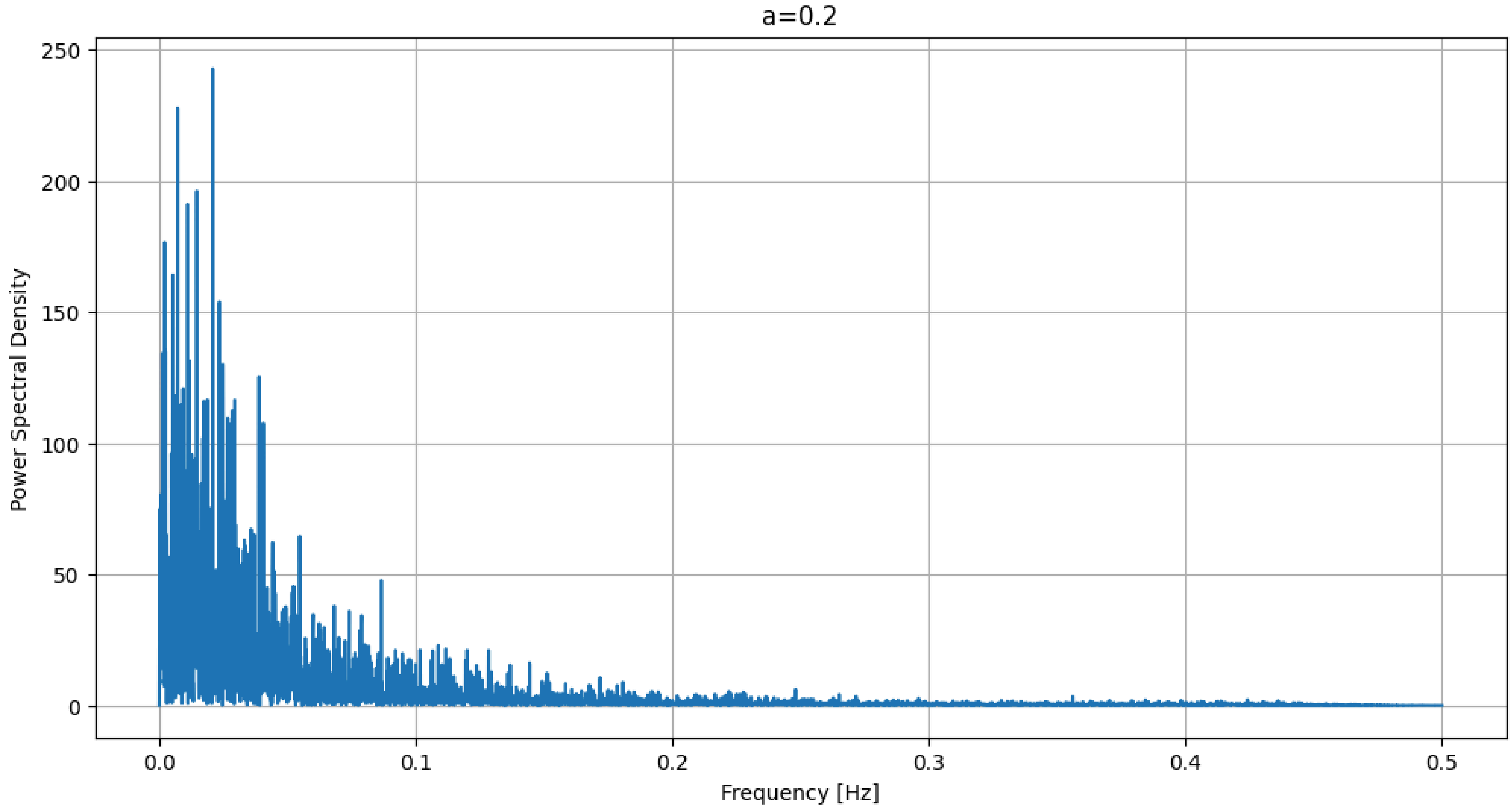
In [ ]:
```
def Sth_y(f, a):
    numerator = 1 + np.exp(-20*a) * (np.exp(2) - 2*np.cos(40*np.pi*f))
    denominator = a**2 + 4*np.pi**2*f**2
    return (numerator/denominator) * (f < 0.5)
```

# 4

In [ ]:
```
#4
t_vals = list(range(0,(2**(13*T))-T+1))
Y_t = generate_Y(t_vals, delta_t, m, X_k, 0)

frequencies, S_y1 = periodogram(Y_t, fs=1/T)
plt.figure(figsize=(12, 6))
plt.plot(frequencies, S_y1)
plt.xlabel("Frequency [Hz]")
plt.ylabel("Power Spectral Density")
plt.title("a=0")
plt.grid()
plt.show()
```




In [ ]:
```
Y_t_2 = generate_Y(t_vals, delta_t, m, X_k, 0.2)
frequencies_2, S_y2 = periodogram(Y_t_2, fs=1/T)
plt.figure(figsize=(12, 6))
plt.plot(frequencies_2, S_y2)
plt.xlabel("Frequency [Hz]")
plt.ylabel("Power Spectral Density")
plt.title("a=0.2")
plt.grid()
plt.show()
```



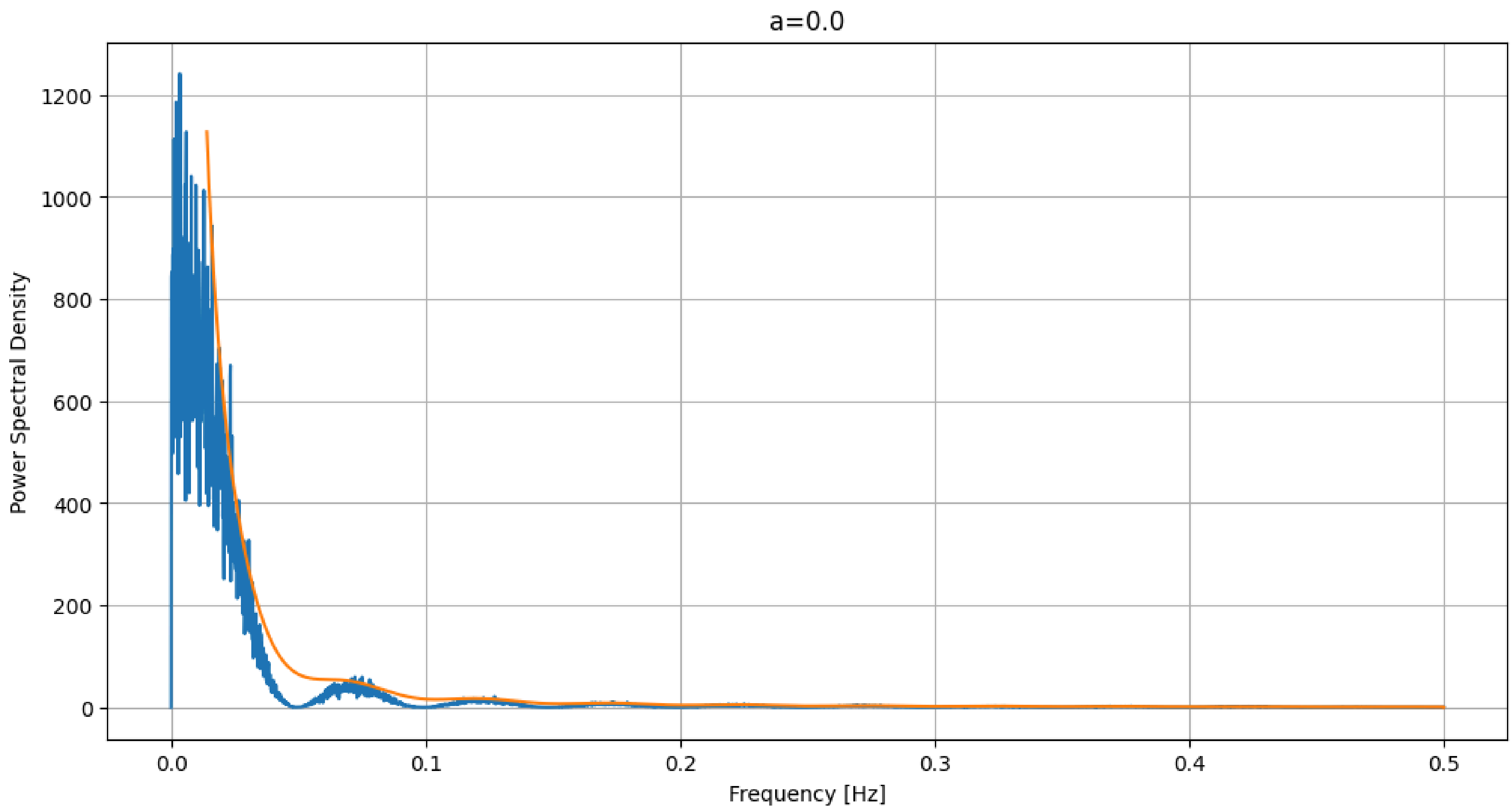

In [ ]:
```
print(len(frequencies))
print(len(S_y1))
```

```
4097
4097
```

# 5

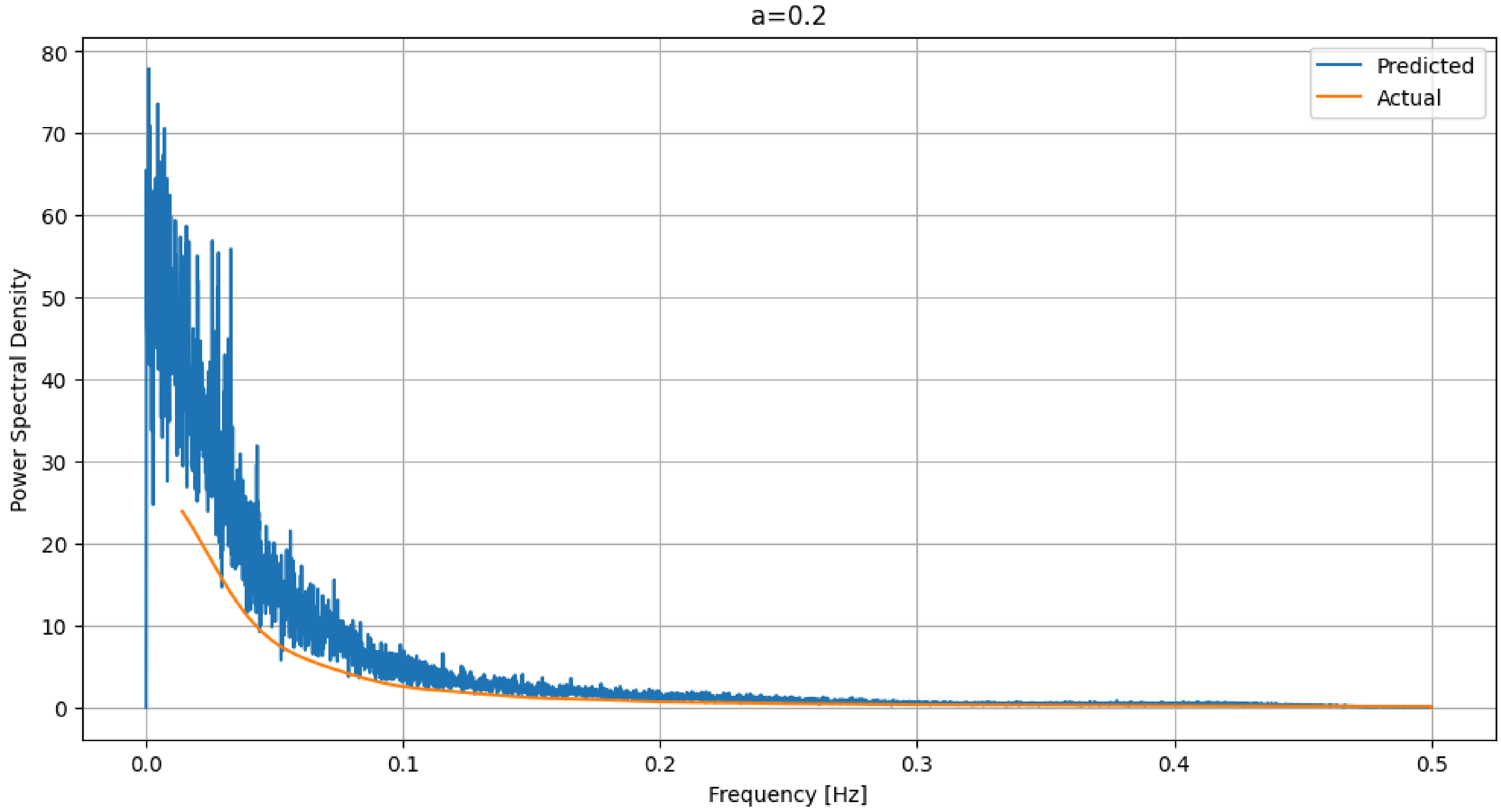
In [ ]:
```
#5
t_vals = list(range(0,(2**(13*T))-T+1))
N = 20
#Case 1: a = 0
S_Y_avg = np.zeros(4097)
freq = []
for i in range(N):
  X_k = np.random.normal(mean, var, 10000)
  Y = generate_Y(t_vals, delta_t, m, X_k, 0)
  frequencies, S_y = periodogram(Y, fs=1/T)
  if i == 0:
    freq = frequencies
  S_Y_avg += np.array(S_y)

S_Y_avg = S_Y_avg/N
S_Y = []
for f in freq:
  S_Y.append(Sth_y(f, 0))

#We need to eliminate f around 0, which can make S_Y(f) becomes undefined.
first = np.where(freq > 0.014)[0][0]
plt.figure(figsize=(12, 6))
plt.plot(freq, S_Y_avg, label="Predicted")
plt.plot(freq[first:], S_Y[first:], label="Actual")
plt.xlabel("Frequency [Hz]")
plt.ylabel("Power Spectral Density")
plt.title("a=0.0")
plt.grid()
plt.legend()
plt.show()
```

```
C:\Users\Nitro\AppData\Local\Temp\ipykernel_33172\1086699757.py:6: RuntimeWarning: divide by zero encountered in double_scalars
  return (numerator/denominator) * (f < 0.5)
```




Note: Orange is the true PSD and Blue is the averaged estimated PSD.

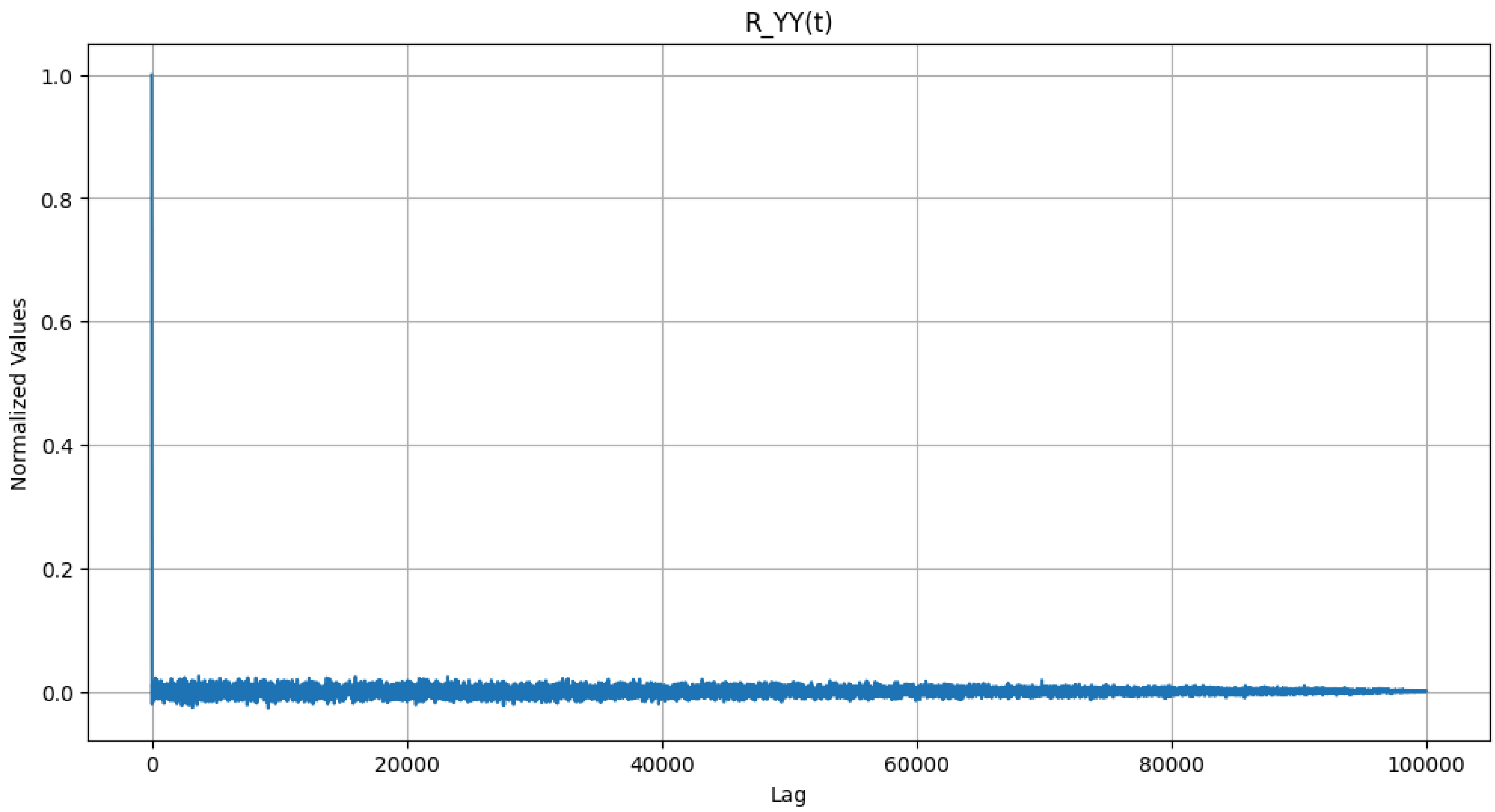
In [ ]:
```
#Case 2: a = 0.2
S_Y_avg = np.zeros(4097)
freq = []
for i in range(N):
  X_k = np.random.normal(mean, var, 10000)
  Y = generate_Y(t_vals, delta_t, m, X_k, 0.2)
  frequencies, S_y = periodogram(Y, fs=1/T)
  if i == 0:
    freq = frequencies
  S_Y_avg += np.array(S_y)

S_Y_avg = S_Y_avg/N
S_Y = []
for f in freq:
  S_Y.append(Sth_y(f, 0.2))

first = np.where(freq > 0.014)[0][0]

plt.figure(figsize=(12, 6))
plt.plot(freq, S_Y_avg, label="Predicted")
plt.plot(freq[first:], S_Y[first:], label="Actual")
plt.xlabel("Frequency [Hz]")
plt.ylabel("Power Spectral Density")
plt.title("a=0.2")
plt.grid()
plt.legend()
plt.show()
```




Explanation: As n increases, the averaged PSD gets closer to the true PSD, $S_Y(f)$, from question 3. Thus, it gives us a better approximation.

# 6

In [ ]:
```
#6
t_vals = list(range(0,100001*T))
X_k = np.random.normal(mean, var, 120000)
Y = generate_Y(t_vals, delta_t, m, X_k, 0.2)
acfY = acf(Y, nlags=len(Y)-1)

plt.figure(figsize=(12, 6))
plt.plot(t_vals, acfY)
plt.xlabel("Lag")
plt.ylabel("Normalized Values")
plt.title("R_YY(t)")
plt.grid()
plt.show()
```




In [ ]:
```
S_y = abs(np.fft.fftshift(np.fft.fft(acfY)))
n = len(S_y)
freq = np.fft.fftshift(np.fft.fftfreq(n, 1/T))

S_Y = []
for f in freq:
```

```
  S_Y.append(Sth_y(f, 0.2))

plt.figure(figsize=(12, 6))
plt.plot(freq[int(n/2):], S_y[int(n/2):], label="predicted")
plt.plot(freq[int(n/2):], S_Y[int(n/2):], label="actual")
plt.xlabel("Frequency [Hz]")
plt.ylabel("Power Spectral Density")
plt.title("a=0.2")
plt.grid()
plt.legend()
plt.show()
```