

APPLIED PARALLEL PROGRAMMING

PARALLELIZE THE FAST FOURIER TRANSFORM

Group 8

Welcome To Our Presentation



Member 1



Member 2



Nguyễn Sỹ Phong
20120159

Trần Thái Vũ
20120632

Group members

Sections

I. LOADING DATA

II. TARGET

III. Smoothing by mean

IV. PARALLELIZATION SMOOTHING BY MEAN

V. FAST FOURIER TRANSFORM

Let's get to know about this project

||



Project Description

Biên đổi Fourier nhanh (FFT), một thuật toán nền tảng được sử dụng rộng rãi trong xử lý tín hiệu, xử lý hình ảnh và nhận dạng giọng nói. Ở đây, nhóm triển khai thuật toán trên nền tảng lập trình CUDA để tăng tốc độ xử lý dữ liệu. FFT là một thuật toán quan trọng trong xử lý dữ liệu, được sử dụng để phân tích và tổng hợp tín hiệu theo miền tần số.

Thuật toán FFT hoạt động trên một chuỗi các điểm dữ liệu để chuyển đổi nó từ miền thời gian sang miền tần số và ngược lại. Việc chuyển đổi này rất quan trọng để phân tích các thành phần tần số trong tín hiệu, nén hình ảnh hoặc nhận dạng các mẫu trong dữ liệu giọng nói.

Ở đây nhóm lấy dữ liệu từ cuộc thi mang tên “VSB Power Line Fault Detection”. Và các bước thực hiện sẽ được thể hiện chi tiết qua các segments.

FAST FOURIER TRANSFORM - DENOISING

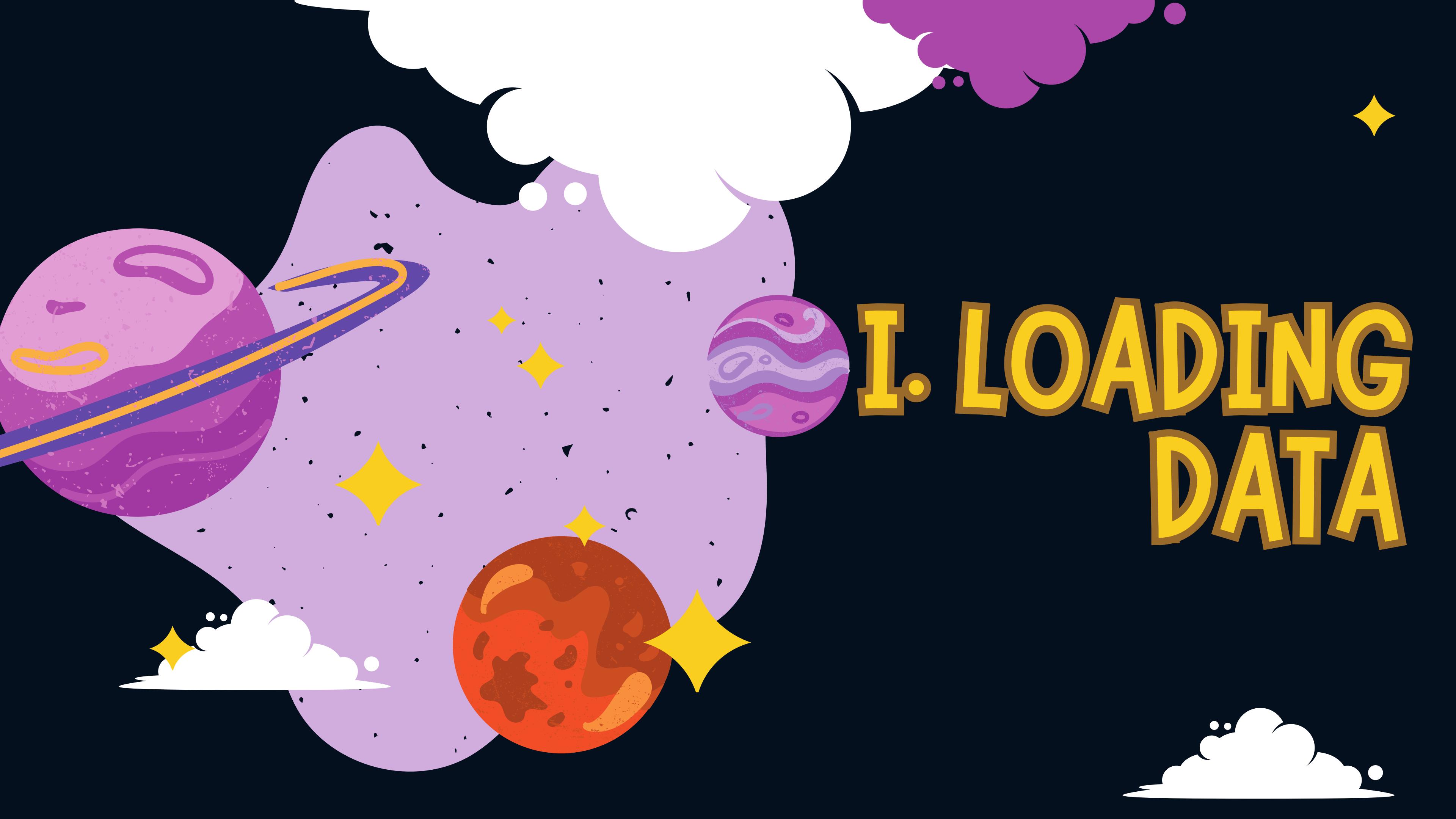
Mô tả về dữ liệu:

- **Problem:** xử lý các sự cố trong các đường dây truyền tải điện, dẫn đến hiện tượng phóng điện một phần. Nếu không được giải quyết, phóng điện một phần có thể gây hại nghiêm trọng cho thiết bị, có thể dẫn đến hỏng hoàn toàn. Thách thức là phát hiện các phóng điện một phần sớm, cho phép sửa chữa kịp thời và ngăn chặn hư hại nghiêm trọng.
- **Signal data:** mỗi tín hiệu được ghi lại bao gồm 800,000 lần đo điện áp được thực hiện từ một đường dây điện trong thời gian 20 mili giây. Vì lưới điện hoạt động với tần số 50 Hz, thời gian này bao gồm một chu kỳ hoàn chỉnh của lưới điện. Dữ liệu cũng phản ánh hệ thống điện 3 pha, có nghĩa là cả ba pha đều được giám sát đồng thời.

FAST FOURIER TRANSFORM - DENOISING

Mô tả về features:

- **metadata[train/test].csv:**
 - **idmeasurement:** mã ID cho bộ ba tín hiệu được ghi cùng lúc.
 - **signalid:** định danh duy nhất cho mỗi tín hiệu, phân biệt giữa bộ dữ liệu train và test. Các ID là tuần tự nhưng riêng biệt cho train và test (bắt đầu từ '0' cho train và '8712' cho test).
 - **phase:** mã ID pha trong bộ ba tín hiệu. Các pha có thể bị ảnh hưởng hoặc không bởi sự cố trên đường dây.
 - **target:** Chỉ ra điều kiện của pha '0' nếu không có lỗi và '1' nếu phát hiện lỗi trong đường dây.
- **[train/test].parquet:** lưu trữ dữ liệu tín hiệu thực tế. Mỗi cột trong tệp Parquet đại diện cho một tín hiệu chứa 800,000 lần đo, được lưu trữ dưới dạng kiểu dữ liệu int8. Tệp Parquet được cấu trúc để tải dữ liệu lớn một cách hiệu quả, đặc biệt hữu ích cho việc truy cập một phần dữ liệu mà không cần tải toàn bộ tệp vào bộ nhớ.



I. LOADING DATA

```
[33] 0s
import numpy as np
import pandas as pd
import seaborn as sns
from numpy.fft import *
import pyarrow.parquet as pq
import matplotlib.pyplot as plt

sns.set_style("whitegrid")
```



Import những thư viện cần thiết

```
[34] signals = pq.read_table('/content/drive/MyDrive/Colab Notebooks/train.parquet', columns=[str(i) for i in range(999)]).to_pandas()
signals
```

0	1	2	3	4	5	6	7	8	9	...	989	990	991	992	993	994	995	996	997	998	
0	18	1	-19	-16	-5	19	-15	15	-1	-16	...	4	-18	10	9	18	-20	1	18	-19	-6
1	18	0	-19	-17	-6	19	-17	16	0	-15	...	1	-20	8	8	20	-19	2	18	-18	-6
2	17	-1	-20	-17	-6	19	-17	15	-3	-15	...	1	-20	6	6	17	-22	0	18	-18	-6
3	18	1	-19	-16	-5	20	-16	16	0	-15	...	3	-21	5	6	18	-19	1	18	-18	-6
4	18	0	-19	-16	-5	20	-17	16	-2	-14	...	3	-20	5	6	19	-21	1	19	-18	-5
...	
799995	19	2	-18	-15	-4	21	-16	16	-1	-17	...	-1	-21	6	6	16	-19	1	20	-17	-5
799996	19	1	-19	-15	-4	20	-17	15	-3	-18	...	1	-21	6	6	15	-21	1	20	-18	-5
799997	17	0	-19	-15	-4	21	-16	14	-2	-18	...	4	-19	7	8	16	-19	1	19	-17	-5
799998	19	1	-18	-14	-3	22	-16	17	-1	-17	...	0	-19	8	8	16	-19	1	20	-18	-6
799999	17	0	-19	-14	-4	21	-17	14	-4	-15	...	1	-18	9	9	14	-21	0	19	-17	-5

800000 rows × 999 columns



I.1 Tải dữ liệu từ file train.parquet vào biến signals dưới dạng DataFrame của Pandas.

- **Hàm pq.read_table:** đọc dữ liệu từ file Parquet (Parquet là một định dạng lưu trữ cột hiệu quả cho Hadoop, thường được sử dụng trong xử lý dữ liệu lớn do khả năng nén cao và đọc/ghi nhanh).
- **Tham số columns=[str(i) for i in range(999)]:** chỉ định danh sách các cột để đọc vào, từ cột '0' đến cột '998'.
- **.to_pandas():** chuyển đổi kết quả đọc được thành DataFrame Pandas.

```
[35] signals = np.array(signals).T.reshape((999//3, 3, 800000))
signals

array([[[ 18,  18,  17, ...,  17,  19,  17],
       [  1,   0, -1, ...,  0,   1,   0],
       [-19, -19, -20, ..., -19, -18, -19]],

      [[-16, -17, -17, ..., -15, -14, -14],
       [ -5,  -6,  -6, ...,  -4,  -3,  -4],
       [ 19,  19,  19, ...,  21,  22,  21]],

      [[-15, -17, -17, ..., -16, -16, -17],
       [ 15,  16,  15, ...,  14,  17,  14],
       [ -1,   0,  -3, ...,  -2,  -1,  -4]],

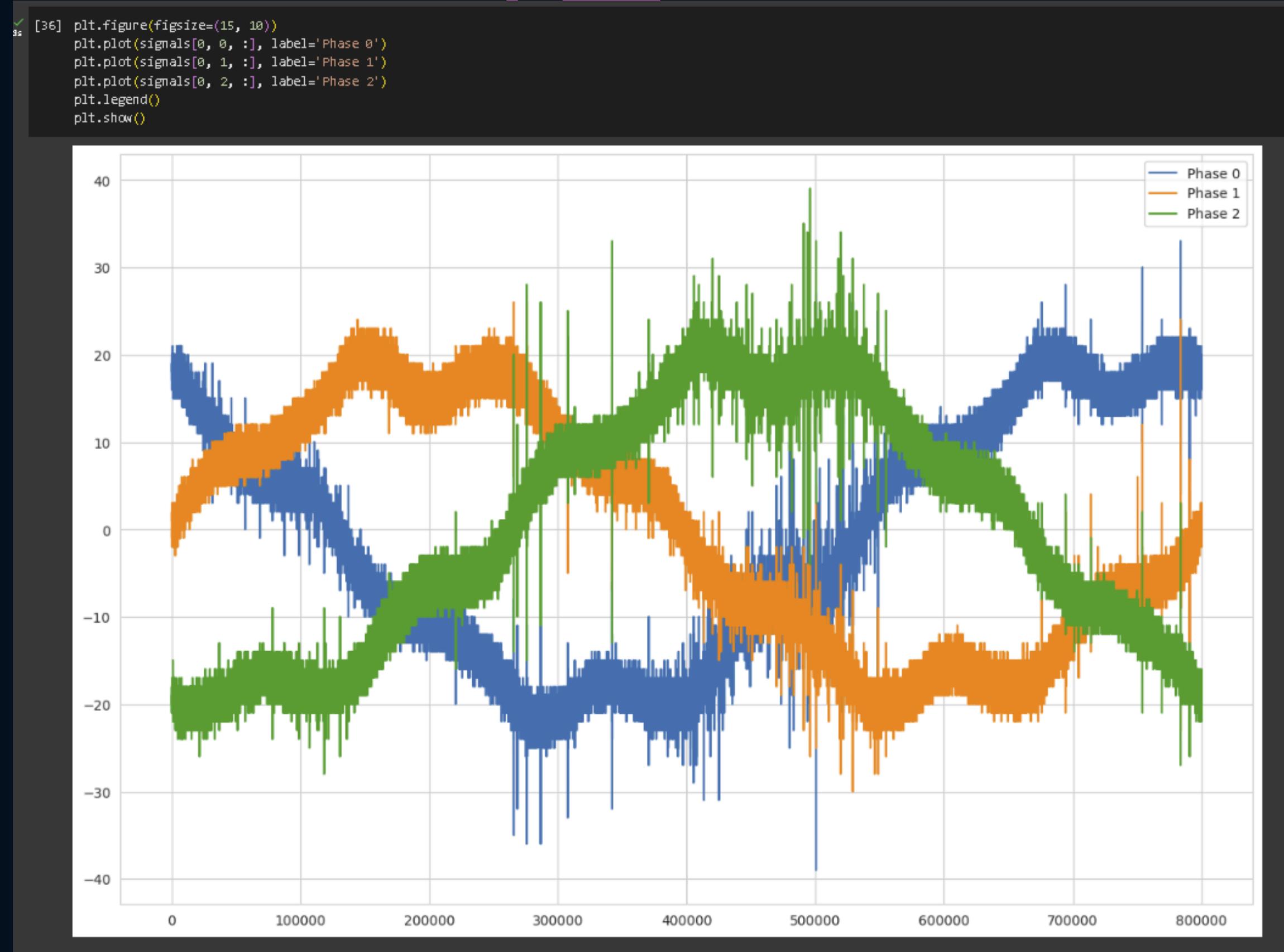
      ...,
      [[-18, -20, -20, ..., -19, -19, -18],
       [ 10,   8,   6, ...,   7,   8,   9],
       [  9,   8,   6, ...,   8,   8,   9]],

      [[ 18,  20,  17, ...,  16,  16,  14],
       [-20, -19, -22, ..., -19, -19, -21],
       [  1,   2,   0, ...,   1,   1,   0]],

      [[ 18,  18,  18, ...,  19,  20,  19],
       [-19, -18, -18, ..., -17, -18, -17],
       [ -6,  -6,  -6, ...,  -5,  -6,  -5]]], dtype=int8)
```

I.2. Reshape dữ liệu tín hiệu để phù hợp với việc phân tích. Mỗi tín hiệu được ghi lại trong một cột của DataFrame, và mỗi tín hiệu chứa 800,000 điểm dữ liệu.

- **np.array(signals):** Chuyển DataFrame signals thành một mảng NumPy.
- **.T:** Chuyển vị (transpose) mảng để cột trở thành hàng và ngược lại. Thay đổi cách dữ liệu được tổ chức từ "cột theo tín hiệu" sang "hàng theo tín hiệu".
- **.reshape((999//3, 3, 800000)):** Dùng để tái cấu trúc mảng thành ba chiều, với kích thước mới là (333, 3, 800000). Mỗi tín hiệu điện áp có 800,000 điểm dữ liệu, và mỗi bộ ba tín hiệu (ứng với ba pha của hệ thống điện) được tổ chức cùng nhau.



I.3. Đồ thị của các tín hiệu.

Vẽ đồ thị cho ba pha tín hiệu từ bộ ba tín hiệu đầu tiên trong mảng đã reshape.

III. TARGET





II.1

**Đọc dữ liệu từ file
metadata_train.csv và
hiển thị một vài dòng
đầu tiên.**

II.2

**Lấy cột 'target' từ
DataFrame.**

II.3

**Đếm số lượng
Target và vẽ biểu
đồ.**

Mục đích chính của bước này là để phân tích và hiểu sự phân bố của các target trong tập dữ liệu train.

II. TARGET

```
[37]: train_df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/metadata_train.csv')
train_df.head()
```

	signal_id	id_measurement	phase	target
0	0	0	0	0
1	1	0	1	0
2	2	0	2	0
3	3	1	0	1
4	4	1	1	1

-----> **II.1. Đọc dữ liệu từ file metadata_train.csv và hiển thị một vài dòng đầu tiên.**

```
[38]: target = train_df['target'][::3]
target
```

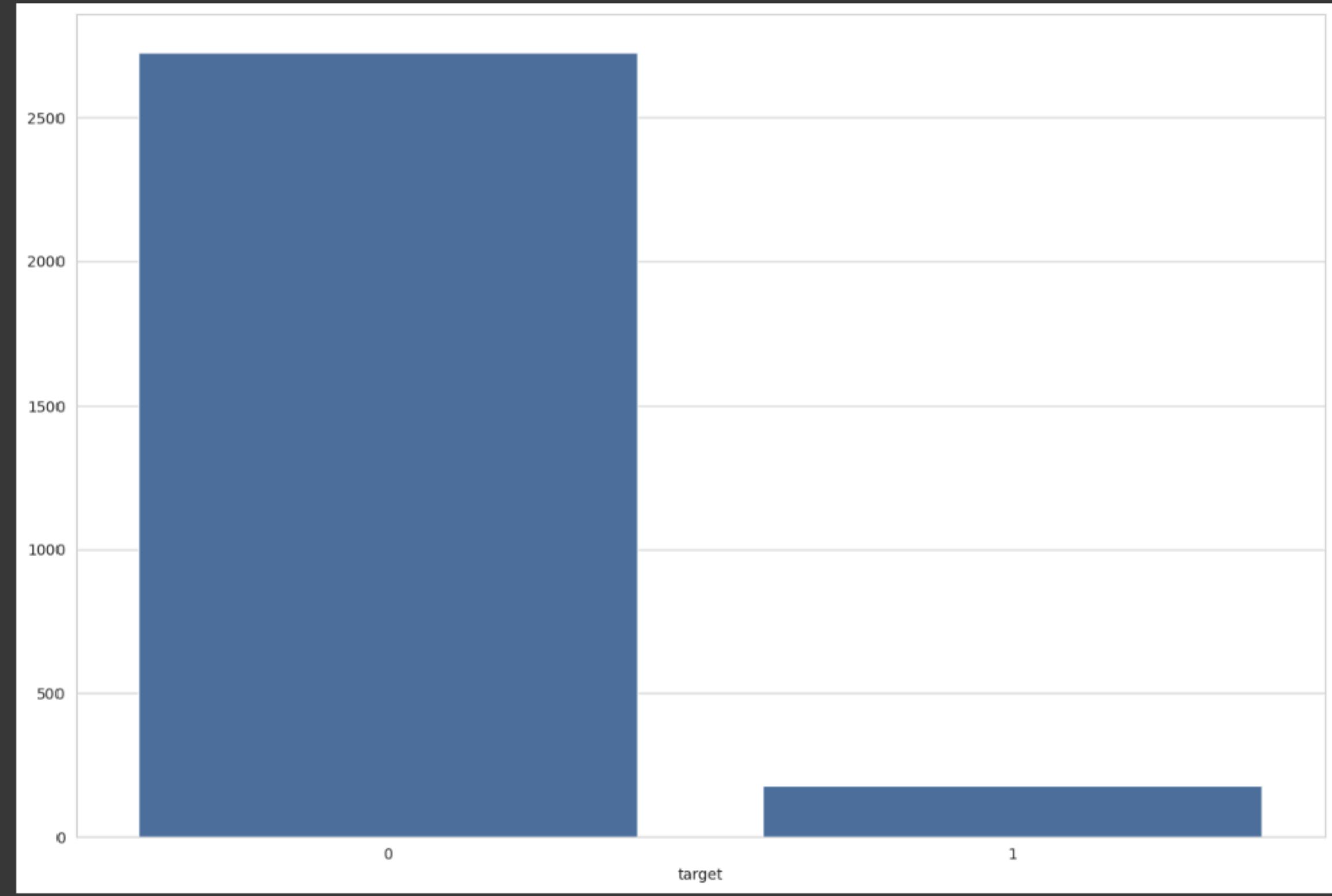
0	0
3	1
6	0
9	0
12	0
..	
8697	0
8700	0
8703	0
8706	0
8709	0

Name: target, Length: 2904, dtype: int64

-----> **II.2 Lấy cột 'target' từ DataFrame.**

Ở đây, nhóm chỉ lấy mỗi giá trị thứ ba (bắt đầu từ chỉ số 0). Mục đích làm giảm số lượng dữ liệu cần xử lý và phân tích.

```
[39] target_count = target.value_counts()
plt.figure(figsize=(15, 10))
sns.barplot(x=target_count.index, y=target_count.values)
plt.show()
```



II.3. Đếm số lượng Target và vẽ biểu đồ.

- valuecounts(): đếm số lượng mỗi giá trị duy nhất trong target, tức là đếm số lượng target là 0 và 1.
- Vẽ biểu đồ cột, trục x là chỉ số (0 và 1, tương ứng với không có lỗi và có lỗi); trục tung là số lượng của mỗi loại.

III. Smoothing by mean



III.1. Mô tả về hàm làm mịn - Smoothing by mean:

Hàm sample: giảm độ phân giải theo thời gian của tín hiệu bằng cách thay thế `kernel_size` điểm liên tiếp bằng trung bình của chúng, làm mịn các biến động ngắn hạn và làm nổi bật các xu hướng lâu dài hơn trong data.

- `def sample(signal, kernel_size):`: định nghĩa hàm với hai tham số, `signal` là mảng NumPy ba chiều chứa tín hiệu cần xử lý và `kernel_size` là số lượng điểm dữ liệu liên tiếp để tính trung bình cho mỗi điểm dữ liệu mới.
- `sampled = np.zeros((signal.shape[0], signal.shape[1], signal.shape[2]//kernel_size))`: khởi tạo một mảng NumPy mới có cùng số hàng và cột như `signal` nhưng có ít điểm dữ liệu hơn theo chiều thứ ba. Số lượng điểm dữ liệu mới này được tính bằng cách lấy tổng số điểm dữ liệu ban đầu (`signal.shape[2]`) chia cho `kernel_size`.

```
[40] def sample(signal, kernel_size):
    sampled = np.zeros((signal.shape[0], signal.shape[1], signal.shape[2]//kernel_size))
    for i in range(signal.shape[2]//kernel_size):
        begin = kernel_size * i
        end = min(kernel_size * (i + 1), signal.shape[2])
        sampled[:, :, i] = np.mean(signal[:, :, begin:end], axis=2)
    return sampled
```

- `for i in range(signal.shape[2]//kernel_size):`: vòng lặp này chạy qua các phân đoạn của mảng `signal` dựa trên `kernel_size`. Với mỗi lần lặp, nó xác định một khối dữ liệu để tính trung bình.
- `begin = kernel_size * i` và `end = min(kernel_size * (i + 1), signal.shape[2])`: đây là các chỉ số bắt đầu và kết thúc của khối data hiện tại mà hàm sẽ tính trung bình. Nếu `kernel_size` là 100, khối đầu tiên sẽ bao gồm các điểm từ chỉ số 0 đến 99, khối tiếp theo từ 100 đến 199, và cứ tiếp tục như vậy cho đến khi kết thúc mảng.
- `sampled[:, :, i] = np.mean(signal[:, :, begin:end], axis=2)`: tính trung bình của `signal` trên khối được xác định (từ `begin` đến `end`) theo chiều thứ ba (`axis=2`) và lưu trữ kết quả vào mảng `sampled` tại vị trí tương ứng.
- `return sampled`: trả về mảng `sampled` chứa tín hiệu đã được làm mịn.

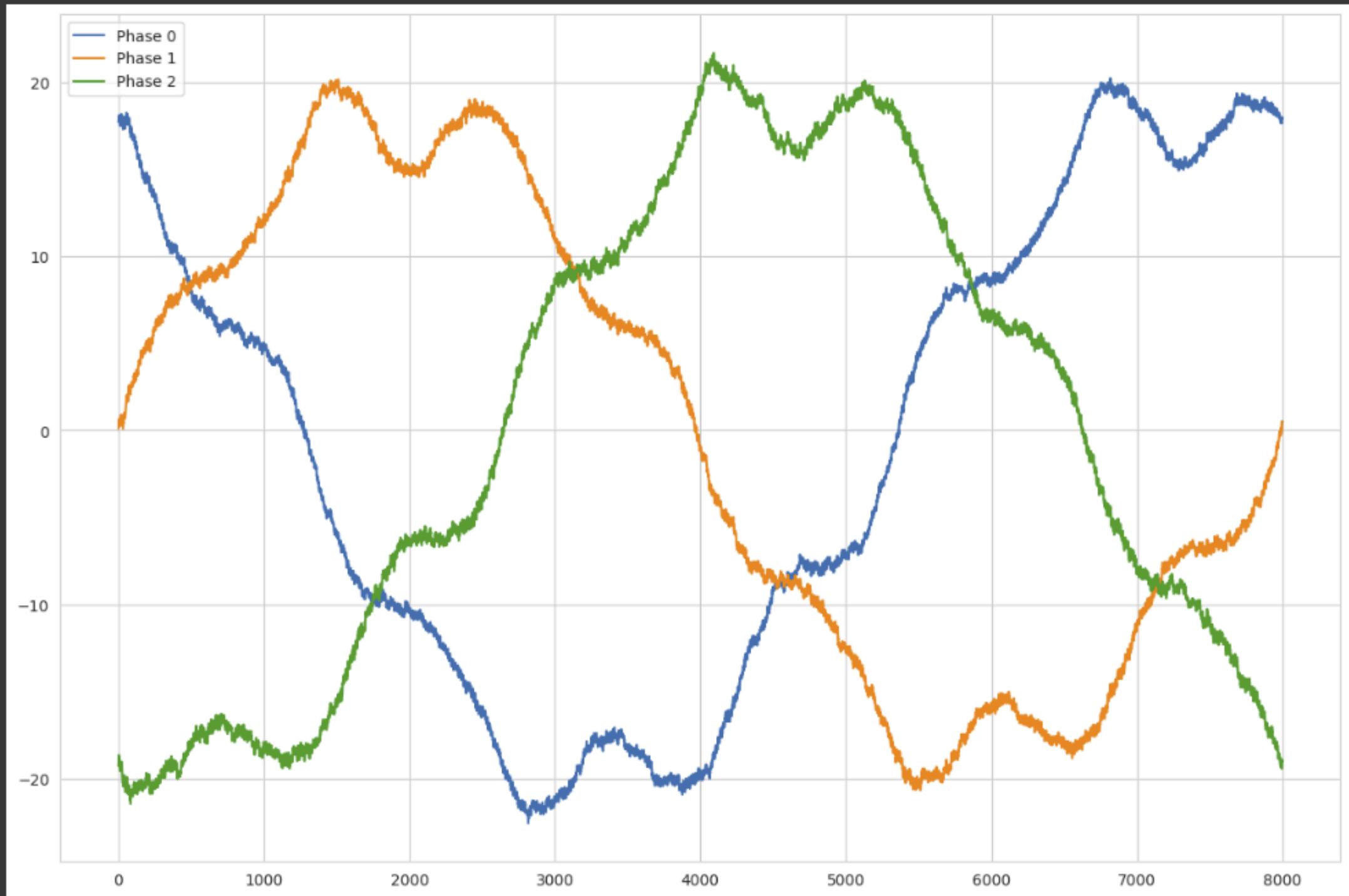
```
[41] sampled = sample(signals, 100)
```

III.2. Áp dụng Smoothing by mean cho signals với kernel size là 100.

Ở bước này:

1. Hàm sample sẽ được gọi với hai mảng signals và 100 làm kích thước kernel cho đường trung bình động.
2. Hàm sẽ tạo một mảng mới sampled, mảng này sẽ có cùng số hàng và cột như signals, nhưng độ dài của chiều thứ ba sẽ bị giảm đi 100 vì đường trung bình động được tính trên cứ 100 điểm của tín hiệu ban đầu.
3. Với mỗi bộ 100 điểm đọc theo chiều thứ ba của signals, hàm sẽ tính giá trị trung bình và lưu trữ ở vị trí tương ứng trong mảng sampled.
4. Cuối cùng, hàm sẽ trả về mảng sampled, hiện chứa tín hiệu đã được làm mịn.

```
[42]: plt.figure(figsize=(15, 10))
plt.plot(sampled[0, 0, :], label='Phase 0')
plt.plot(sampled[0, 1, :], label='Phase 1')
plt.plot(sampled[0, 2, :], label='Phase 2')
plt.legend()
plt.show()
```



III.3. Đồ thị của các tín hiệu sau khi được làm mịn.

Sử dụng thư viện Matplotlib để vẽ biểu đồ các pha của tín hiệu đã được làm mịn.

IV.

PARALLELIZATION

SMOOTHING BY

MEAN



```
[43] from numba import jit, prange
      import time

      @jit(nopython=True, parallel=True)
      def sample_parallel(signal, kernel_size):
          new_length = signal.shape[2] // kernel_size
          sampled = np.zeros((signal.shape[0], signal.shape[1], new_length), dtype=np.float32) # Ensure dtype is float for mean calculation
          for i in prange(new_length):
              begin = kernel_size * i
              end = min(kernel_size * (i + 1), signal.shape[2])
              for j in range(signal.shape[0]):
                  for k in range(signal.shape[1]):
                      # Manually compute the mean to avoid issues with np.mean and datatypes in Numba
                      segment_sum = 0.0
                      for l in range(begin, end):
                          segment_sum += signal[j, k, l]
                      segment_mean = segment_sum / (end - begin)
                      sampled[j, k, i] = segment_mean
          return sampled
```

IV.1. Hàm `sample_parallel` thực hiện việc làm mịn một mảng tín hiệu đa chiều thông qua phương pháp Moving Averages.

- `@jit(nopython=True, parallel=True)`: một decorator từ thư viện Numba, biên dịch hàm sử dụng LLVM compiler framework. `nopython` đảm bảo rằng code chạy mà không cần GIL (Global Interpreter Lock) của Python, và `parallel=True` cho phép tự động song song hóa việc thực thi hàm.
- `def sample_parallel(signal, kernel_size)`: định nghĩa hàm với hai tham số là `signal`, một mảng NumPy chứa dữ liệu tín hiệu và `kernel_size`, một số nguyên biểu diễn kích thước của window mà trên đó tính toán giá trị trung bình.

- `new_length = signal.shape[2] // kernel_size`: tính toán xem có bao nhiêu windows có kích thước `kernel_size` vừa với chiều thứ ba của `signal`.
- `sampled = np.zeros((signal.shape[0], signal.shape[1], new_length), dtype=np.float32)`: khởi tạo một mảng mới NumPy kiểu `float32` để lưu trữ tín hiệu đã được làm mịn, với chiều thứ ba giảm xuống tương ứng với `new_length`.
- `for i in prange(new_length)`: lặp qua chiều dài mới của chiều thứ ba sử dụng `prange`, cho phép thực thi song song vòng lặp.
- Bên trong các vòng lặp lồng nhau qua `j` và `k`, lặp qua hai chiều đầu tiên của `signal` và ở đây:
 - Một biến tạm thời `segment_sum` được sử dụng để tính tổng các giá trị tín hiệu trong window hiện tại được định nghĩa bởi `begin` và `end`.
 - Trung bình của window hiện tại được tính toán thủ công bằng cách chia `segment_sum` cho số lượng phần tử trong window (được cho bởi `(end - begin)`).
 - Giá trị trung bình tính toán được gán vào vị trí tương ứng trong mảng `sampled`.
- `return sampled`: trả về mảng tín hiệu đã được làm mịn `sampled`.

```
[44] # Đo thời gian cho hàm sample
ss start_time_sample = time.time()
sampled = sample(signals, 100)
end_time_sample = time.time()
total_time_sample = end_time_sample - start_time_sample

print(f"Thời gian thực thi của hàm sample: {total_time_sample:.2f} giây")

# Đo thời gian cho hàm sample_parallel
# Note: Cần "Warm up" cho hàm sample_parallel vì là lần đầu tiên chạy để tránh tính thời gian biên dịch numba vào thời gian thực thi.
sample_parallel(signals, 100) # "Warm up" for Numba compilation

start_time_parallel = time.time()
sampled_parallel = sample_parallel(signals, 100)
end_time_parallel = time.time()
total_time_parallel = end_time_parallel - start_time_parallel

print(f"Thời gian thực thi của hàm sample_parallel: {total_time_parallel:.2f} giây")

Thời gian thực thi của hàm sample: 2.58 giây
Thời gian thực thi của hàm sample_parallel: 0.90 giây
```

IV.2. So sánh thời gian thực thi của hai hàm xử lý tín hiệu.

V. FAST FOURIER TRANSFORM





Biên đổi Fourier của tín hiệu 1D có độ dài n như sau:

$$f_j = \sum_{k=0}^{n-1} x_k e^{\frac{2\pi i}{n} j k}, \quad \forall j = 0, \dots, n-1$$

Ý tưởng là biểu diễn tín hiệu trong không gian phức tạp. Nó đại khái là tổng của các hàm hình sin. Và có một hệ số cho mỗi tần số có trong tín hiệu.

Tần số nhận các giá trị sau:

- $f = \frac{1}{dn}[0, 1, \dots, \frac{n}{2} - 1, -\frac{n}{2}, \dots, -1]$ if n is even
- $f = \frac{1}{dn}[0, 1, \dots, \frac{n-1}{2}, -\frac{n-1}{2}, \dots, -1]$ if n is odd

Denoising algorithm - Các bước khử nhiễu:

- Áp dụng FFT cho tín hiệu
- Tính tần số liên quan đến từng hệ số
- Chỉ giữ lại các hệ số có tần số đủ thấp (tuyệt đối)
- Tính toán FFT nghịch đảo

V.1. Hàm lọc nhiễu FFT.

Hàm `filtersignal` được định nghĩa để lọc tín hiệu bằng cách sử dụng biến đổi Fourier nhanh (FFT) và loại bỏ các thành phần tần số cao hơn ngưỡng được chỉ định. Cụ thể:

- `signal`: tín hiệu đầu vào cần được lọc.
- `threshold=1e8`: ngưỡng tần số để lọc các thành phần tần số cao. Ngưỡng này được đặt mặc định là (1×10^8), nhưng có thể được chỉ định một giá trị khác khi gọi hàm.
- `fourier = rfft(signal)`: sử dụng hàm `rfft` từ thư viện Numpy để tính biến đổi Fourier nhanh của tín hiệu đầu vào. Hàm `rfft` được sử dụng để xử lý tín hiệu thực và trả về nửa đầu của biến đổi Fourier, bởi vì biến đổi Fourier của một tín hiệu thực là đối xứng.
- `frequencies = rfftfreq(signal.size, d=20e-3/signal.size)`: sử dụng hàm `rfftfreq` để tính mảng các giá trị tần số cho các thành phần tín hiệu trong kết quả `rfft`. Tham số `d` là khoảng thời gian lấy mẫu, được tính bằng cách chia 20 mili giây (được thể hiện bởi `20e-3`) cho kích thước của

```
[15] def filter_signal(signal, threshold=1e8):
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)
```

- `fourier[frequencies > threshold] = 0`: các thành phần trong biến đổi Fourier có tần số cao hơn ngưỡng sẽ được đặt bằng 0. Điều này loại bỏ các thành phần tần số cao, vốn thường gắn liền với nhiễu, từ tín hiệu.
- `return irfft(fourier)`: sử dụng hàm `irfft` để thực hiện biến đổi Fourier nghịch đảo, chuyển tín hiệu đã lọc trở lại miền thời gian. Kết quả là tín hiệu thời gian đã được làm mịn, với nhiều tần số cao đã được loại bỏ.

Hàm `filtersignal` này cho phép làm mịn tín hiệu bằng cách loại bỏ các thành phần tần số không mong muốn, sử dụng biến đổi Fourier. Điều này rất hữu ích trong các ứng dụng xử lý tín hiệu nơi nhiều tần số cao cần được giảm thiểu để cải thiện chất lượng hoặc tính rõ ràng của tín hiệu gốc.

```
[14] from numpy.fft import rfft, irfft, rfftfreq
     from numba import njit

     # Numba functions to filter frequencies based on threshold
     @njit
     def filter_frequencies(fourier, frequencies, threshold):
         for i in range(fourier.size):
             if frequencies[i] > threshold:
                 fourier[i] = 0
         return fourier

     @njit
     def filter_signal_parallel(fourier, frequencies, threshold=1e8):
         return filter_frequencies(fourier, frequencies, threshold)

     # Load the signal data
     parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
     signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
     signals = np.array(signals_df).T.reshape((999//3, 3, 800000)) # Reshape to 3 phases

     # Select the first phase of the first signal
     signal = signals[0, 0, :]

     # Perform FFT
     fourier = rfft(signal)
     frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
```

V.2. Song song hóa hàm lọc nhiễu FFT.

1. Thêm thư viện cần thiết:

- **numpy.fft**: Mô-đun này cung cấp các hàm để tính Biên đổi Fourier Nhanh (FFT) và ngược lại. Hàm `rfft` tính FFT của một đầu vào có giá trị thực, trả về các thành phần tần số không âm. `irfft` là hàm ngược của `rfft`, được sử dụng để chuyển đổi từ miền tần số trở lại miền thời gian. `rfftfreq` sinh ra các khoảng tần số cho một kích thước FFT đầu ra và khoảng mẫu đã cho.
- **numba**: Là một trình biên dịch Just-in-Time (JIT) chuyển một tập hợp con của mã Python và NumPy thành mã máy tốc độ cao. `njit` là một trình trang trí chỉ ra rằng Numba nên biên dịch hàm này trong chế độ no-Python, nơi hàm được biên dịch sao cho không truy cập vào các đối tượng Python.

2. Định nghĩa các hàm được cải thiện bởi Numba:

- **filterfrequencies:** Hàm này đặt các phần tử trong mảng biến đổi Fourier (fourier) về 0 nếu tần số tương ứng vượt quá một ngưỡng (threshold) nhất định. Điều này thường được thực hiện để loại bỏ nhiễu tần số cao trong tín hiệu.
- **filtersignalparallel:** Hàm bao bọc gọi filterfrequencies. Nó được cấu trúc để có thể tận dụng khả năng xử lý song song của Numba để cải thiện hiệu suất.

3. Load và định hình lại signal data:

- Tải dữ liệu tín hiệu từ một tệp Parquet, đây là định dạng lưu trữ cột. Dữ liệu được tải vào một DataFrame pandas, sau đó chuyển đổi thành một mảng NumPy. Mảng được chuyển vị và định hình lại để tổ chức dữ liệu một cách có cấu trúc phù hợp cho việc xử lý. Ở đây, mỗi bộ ba cột tương ứng với ba pha của một tín hiệu, và mỗi pha chứa 800.000 điểm dữ liệu.

4. Chọn signal và thực hiện FFT:

- **signal:** Chọn pha đầu tiên của bộ tín hiệu đầu tiên.
- **fourier:** Tính biến đổi Fourier của tín hiệu đã chọn. Biến đổi này chuyển đổi tín hiệu miền thời gian thành các thành phần tần số của nó.
- **frequencies:** Tính các giá trị tần số tương ứng với mỗi phần tử trong mảng fourier, dựa trên kích thước tín hiệu và thời gian mẫu của tín hiệu ($d=20e-3$ cho biết tổng thời gian mẫu tín hiệu là 20 mili giây).

Mục đích chung là tải một tập dữ liệu tín hiệu lớn, áp dụng Fast Fourier Transform để chuyển đổi nó thành miền tần số, lọc các tần số trên một ngưỡng nhất định để giảm nhiễu hoặc loại bỏ các thành phần tần số không mong muốn, và có thể sử dụng quá trình này cho phân tích hoặc tái tạo tín hiệu tiếp theo. Nhóm sử dụng Numba nhằm mục đích tối ưu hóa các thao tác này, đặc biệt hữu ích khi xử lý các tập dữ liệu lớn hoặc yêu cầu hiệu suất thời gian thực.

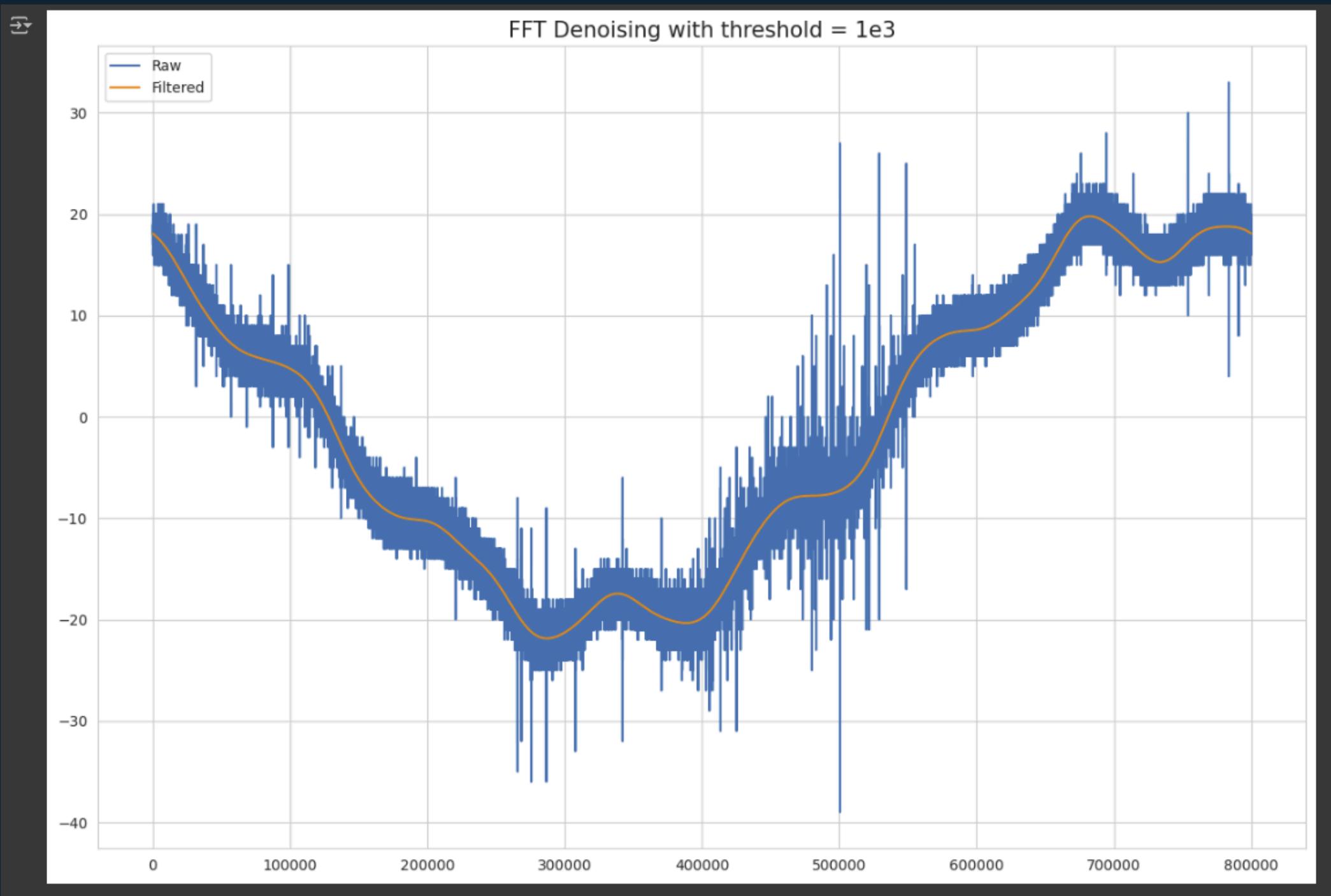
V.3. Thử nghiệm trên các thresholds:

```
[15] # Apply threshold = 1e3
threshold = 1e3 # Lower threshold for testing
filtered_fourier = filter_signal_parallel(fourier, frequencies, threshold)
filtered_signal = irfft(filtered_fourier)

[16] # Plot the original and filtered signals
plt.figure(figsize=(15, 10))
plt.plot(signal, label='Raw')
plt.plot(filtered_signal, label='Filtered')
plt.legend()
plt.title("FFT Denoising with threshold = 1e3", size=15)
plt.show()
```

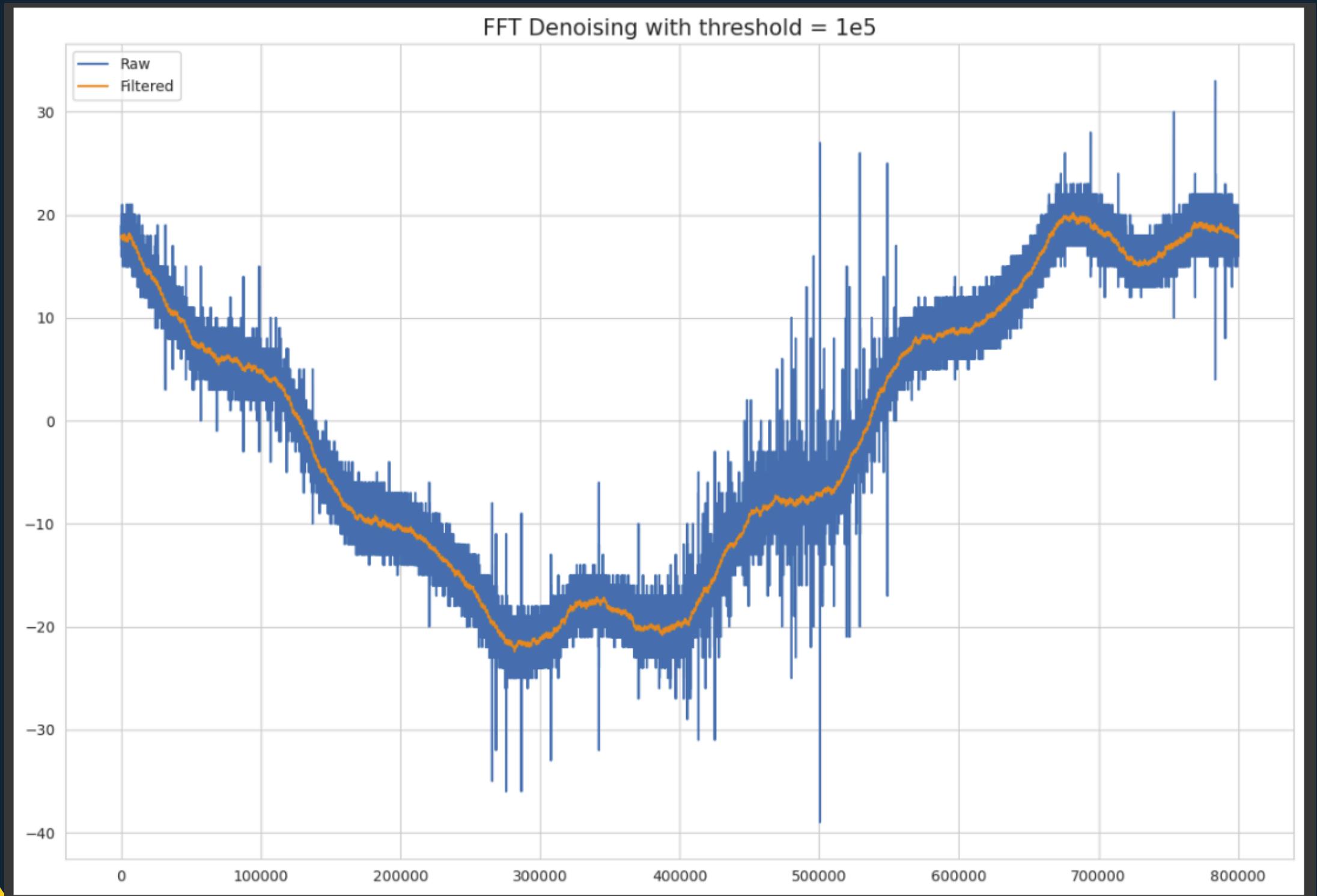
Ý tưởng giải quyết khi thực nghiệm trên một vài Thresholds, và ở đây nhóm test trên các ngưỡng như: 1e3, 1e5, 1e7.

- **Đặt ngưỡng:** ngưỡng được đặt để kiểm soát việc loại bỏ nhiễu. Ngưỡng này cho phép loại bỏ các thành phần tần số mà giá trị nhỏ hơn ngưỡng này, được xem là nhiễu hoặc không quan trọng đối với tín hiệu gốc.
- **Lọc fourier:** `filter_signal_parallel` sử dụng biến đổi Fourier đã tính trước (`fourier`) và tần số (`frequencies`) để áp dụng ngưỡng. Trong hàm này, các thành phần tần số cao hơn ngưỡng được đặt thành không, nhằm giảm nhiễu trong tín hiệu.
- **Biến đổi Fourier ngược:** sau khi các thành phần tần số không mong muốn đã được loại bỏ, `irfft` (Biến đổi Fourier ngược cho tín hiệu thực) được sử dụng để chuyển dữ liệu trở lại từ miền tần số sang miền thời gian, tạo ra tín hiệu đã được lọc.



Hình 1. Giảm nhiễu áp dụng FFT với threshold $1e3$

- **Ngưỡng $1e3$ đã giúp làm giảm đáng kể các nhiễu tần số cao trong khi vẫn giữ được hình dáng chung của tín hiệu.** Tuy nhiên, vẫn có một số biến động nhỏ không được lọc hết, cho thấy ngưỡng này có thể không đủ cao để lọc sạch hoàn toàn nhiễu nhưng vẫn giữ được phần lớn thông tin cần thiết của tín hiệu.
- **Tín hiệu đã lọc mượt mà hơn nhiều so với tín hiệu gốc,** điều này có thể rất hữu ích trong các ứng dụng yêu cầu độ rõ nét cao và giảm thiểu sự can thiệp của nhiễu, như xử lý tín hiệu âm thanh hoặc tín hiệu y tế.

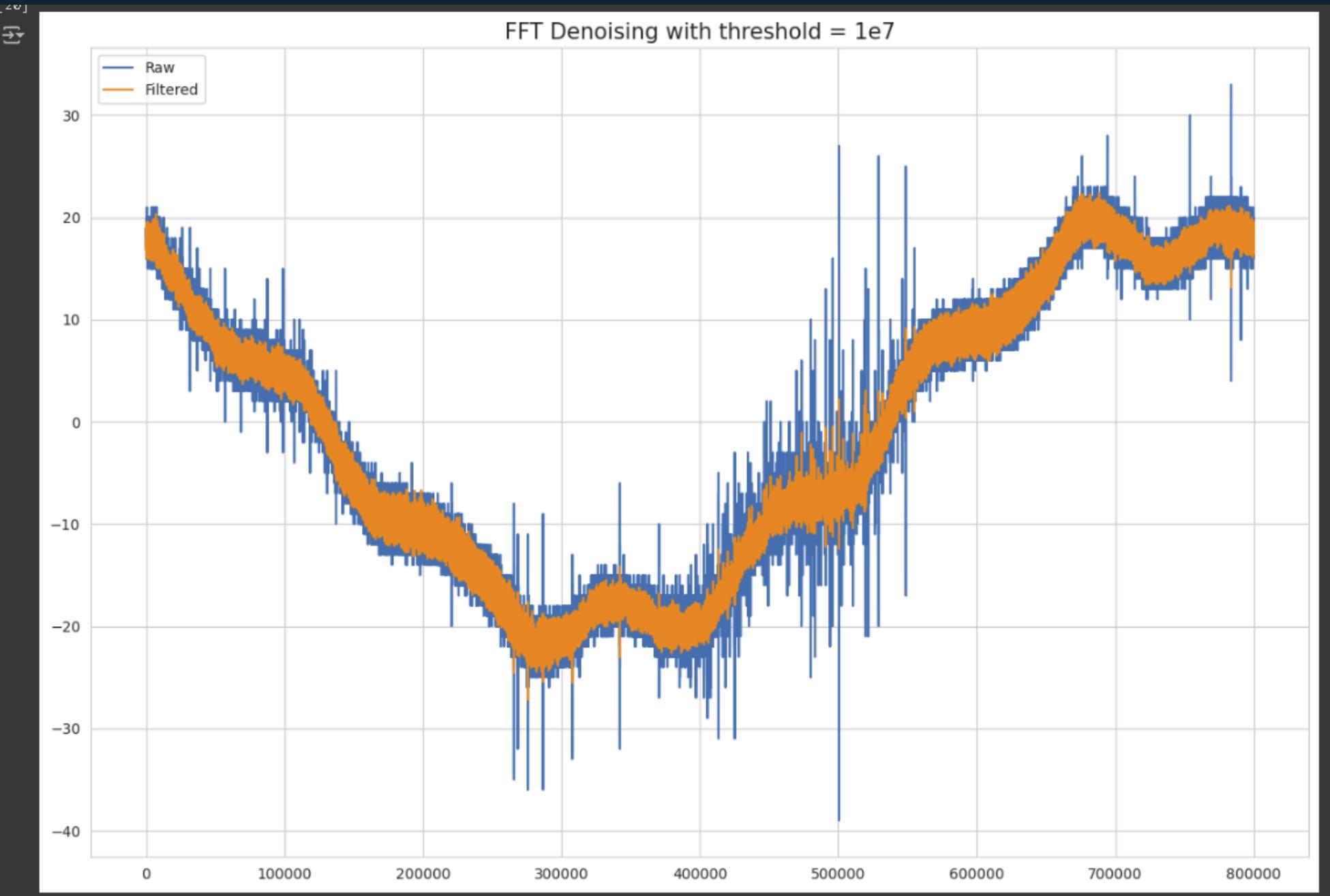


Hình 2. Giảm nhiễu áp dụng FFT với threshold 1e5

```
[17] # Apply threshold 1e5
fourier = rfft(signal) # Recompute the fourier transform to reset it
frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
threshold = 1e5 # Higher threshold for testing
filtered_fourier = filter_signal_parallel(fourier, frequencies, threshold)
filtered_signal = irfft(filtered_fourier)

[18] # Plot the original and filtered signals
plt.figure(figsize=(15, 10))
plt.plot(signal, label='Raw')
plt.plot(filtered_signal, label='Filtered')
plt.legend()
plt.title("FFT Denoising with threshold = 1e5", size=15)
plt.show()
```

- Hiệu quả trong việc loại bỏ nhiễu nhiễu, đặc biệt là các thành phần nhiễu có tần số cao. Tín hiệu đã lọc thể hiện sự ổn định hơn nhiều so với tín hiệu gốc, điều này là lợi ích trong hầu hết các ứng dụng yêu cầu độ rõ nét cao và ít nhiễu.
- Tín hiệu đã lọc vẫn giữ được hình dáng cơ bản của tín hiệu gốc, cho thấy ngưỡng được sử dụng đã loại bỏ nhiễu mà không làm mất đi thông tin quan trọng của tín hiệu. Tuy nhiên, vẫn có thể thấy một số biến động nhỏ trong tín hiệu đã lọc, cho thấy rằng không phải tất cả nhiễu đã được loại bỏ hoàn toàn.

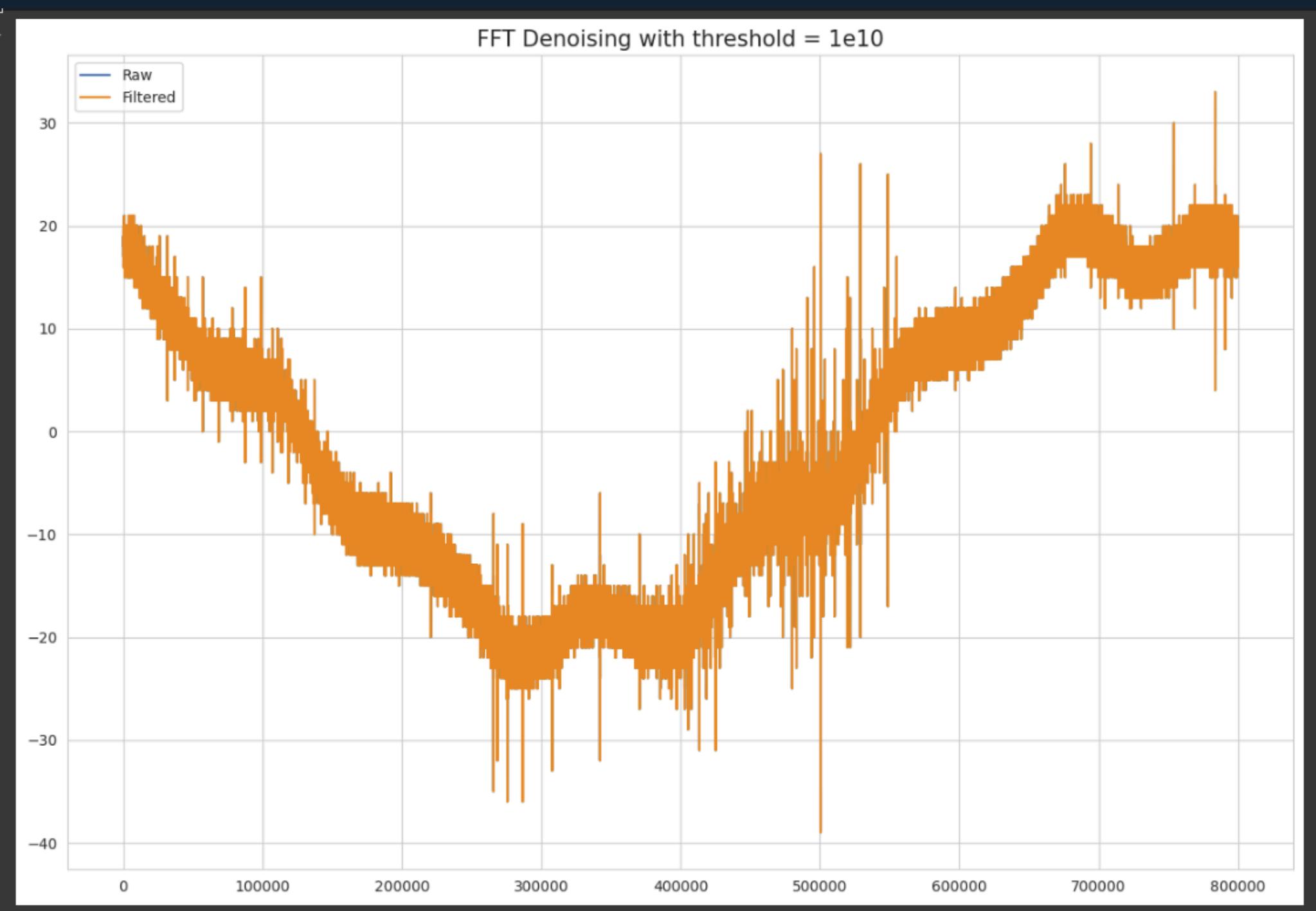


Hình 3. Giảm nhiễu áp dụng FFT với threshold $1e7$

```
[19] # Apply new threshold 1e7
fourier = rfft(signal) # Recompute the fourier transform to reset it
frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
threshold = 1e7 # Higher threshold for testing
filtered_fourier = filter_signal_parallel(fourier, frequencies, threshold)
filtered_signal = irfft(filtered_fourier)

[20] # Plot the original and filtered signals
plt.figure(figsize=(15, 10))
plt.plot(signal, label='Raw')
plt.plot(filtered_signal, label='Filtered')
plt.legend()
plt.title("FFT Denoising with threshold = 1e7", size=15)
plt.show()
```

- **Ta có thể thấy ngưỡng $1e7$ rõ ràng chỉ hiệu quả hơn một ít trong việc loại bỏ nhiễu, giúp tín hiệu đã lọc trở nên mượt mà hơn khá nhiều so với tín hiệu gốc.**
- **Mặc dù tín hiệu đã lọc trơn tru hơn, nhưng cần phải cẩn trọng để đảm bảo việc không loại bỏ quá nhiều thông tin tín hiệu quan trọng. Việc sử dụng một ngưỡng cao như $1e7$ có thể dẫn đến việc một số đặc điểm của tín hiệu bị mất, đặc biệt là trong các tần số cao có chứa thông tin có giá trị.**

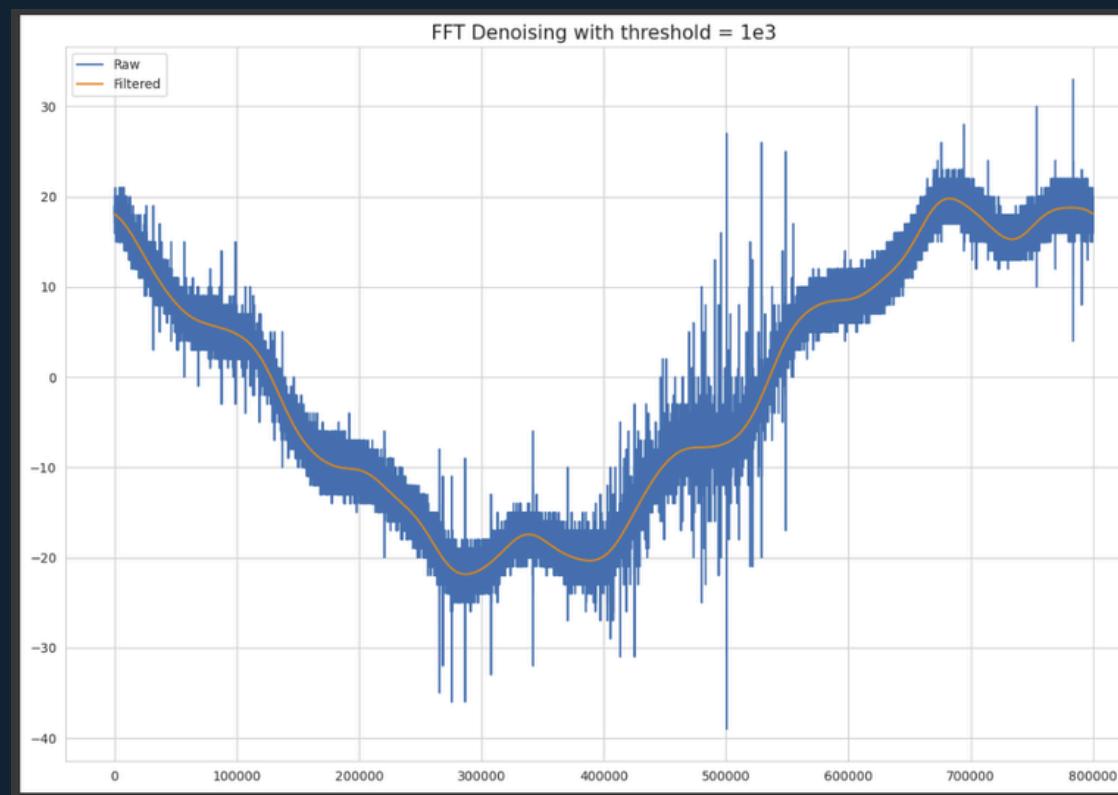


Hình 4. Giảm nhiễu áp dụng FFT với threshold $1e10$

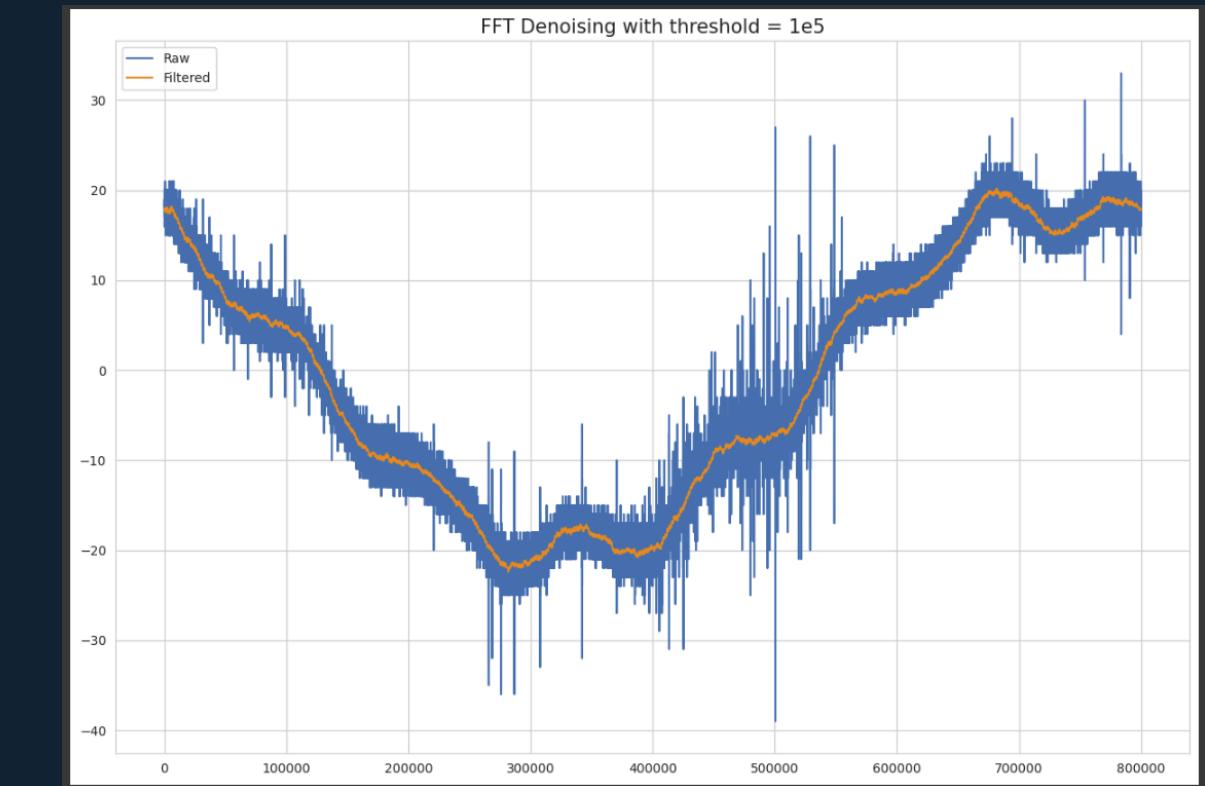
```
[21] # Apply new threshold 1e10
fourier = rfft(signal) # Recompute the fourier transform to reset it
frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
threshold = 1e10 # Higher threshold for testing
filtered_fourier = filter_signal_parallel(fourier, frequencies, threshold)
filtered_signal = irfft(filtered_fourier)

[22] # Plot the original and filtered signals
plt.figure(figsize=(15, 10))
plt.plot(signal, label='Raw')
plt.plot(filtered_signal, label='Filtered')
plt.legend()
plt.title("FFT Denoising with threshold = 1e10", size=15)
plt.show()
```

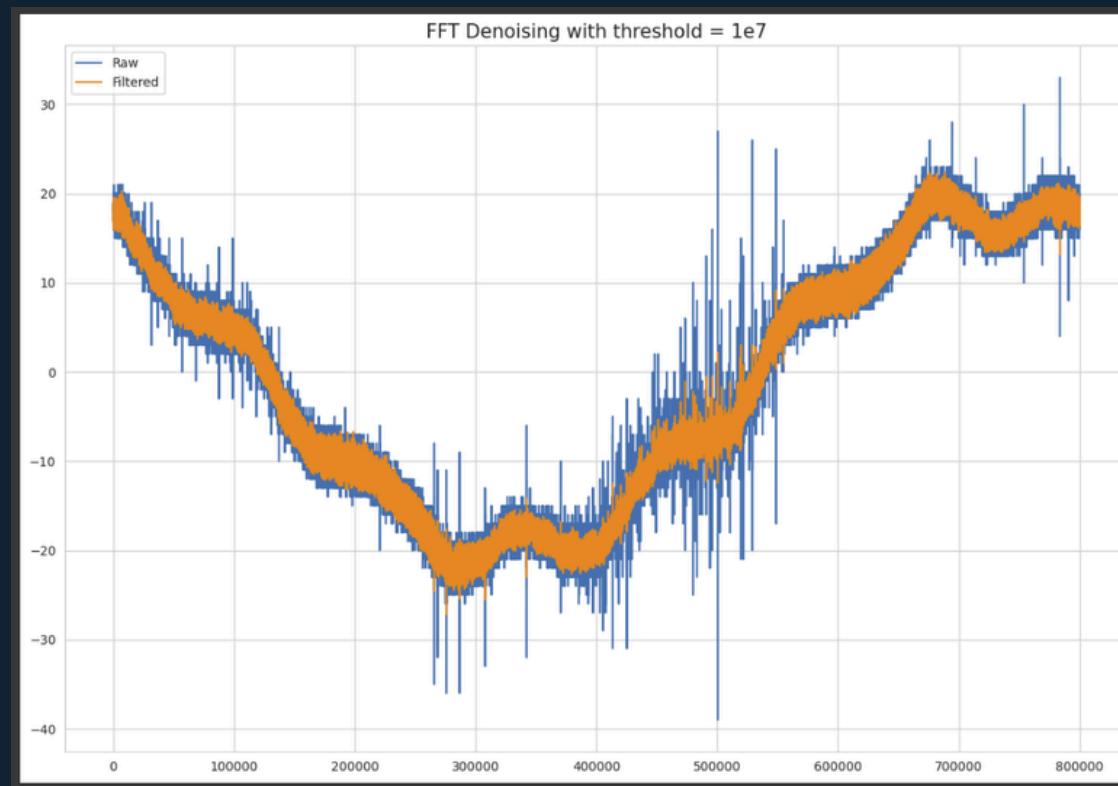
- Qua quan sát ta có thể thấy ngưỡng $1e10$ quá cao, dẫn đến việc loại bỏ gần như toàn bộ các thành phần tần số của tín hiệu. Điều này có thể không còn giữ lại đủ thông tin từ tín hiệu gốc, khiến cho tín hiệu sau khi lọc không còn bất kỳ biến động đáng kể nào so với tín hiệu gốc.
- Sử dụng một ngưỡng cao có thể dẫn đến mất mát các thông tin quan trọng trong tín hiệu, đặc biệt là khi các thông tin này có tần số không cao tới mức ngưỡng đã đặt.



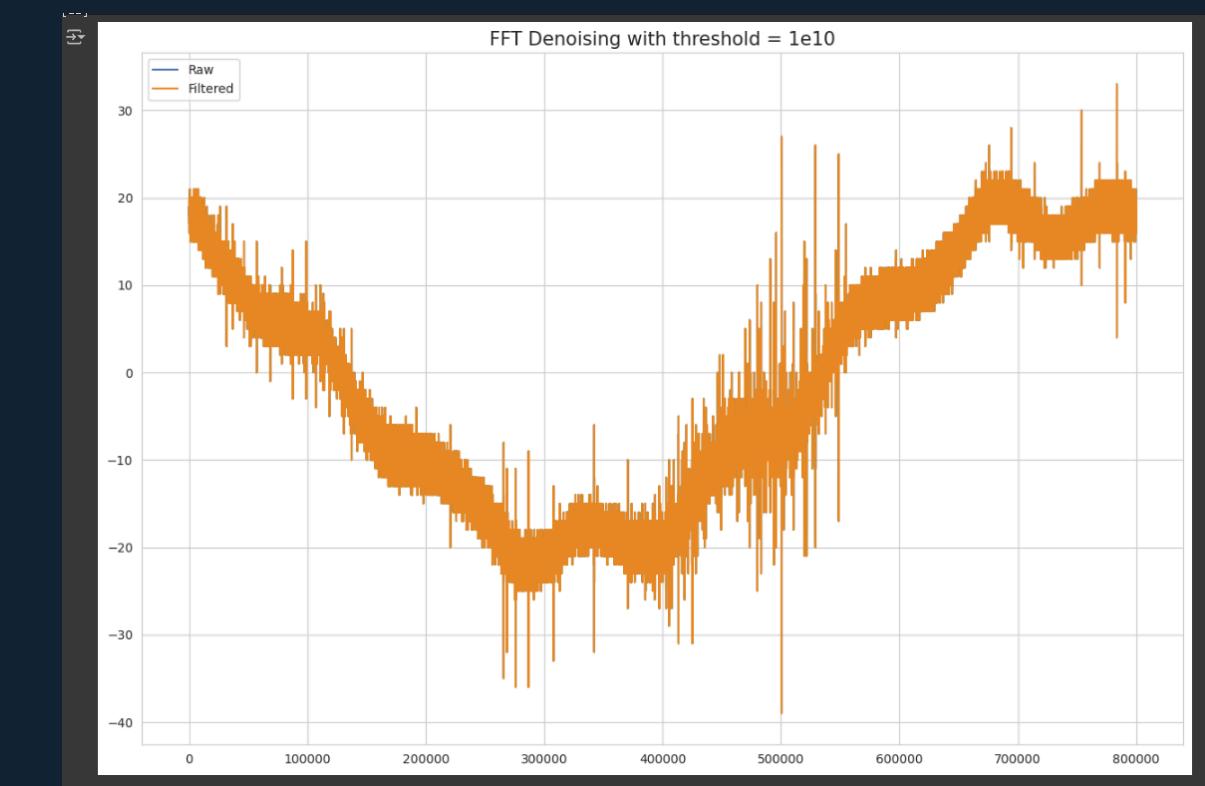
Hình 1. Giảm nhiễu áp dụng FFT với threshold $1e3$



Hình 2. Giảm nhiễu áp dụng FFT với threshold $1e5$



Hình 3. Giảm nhiễu áp dụng FFT với threshold $1e7$



Hình 4. Giảm nhiễu áp dụng FFT với threshold $1e10$

ĐÁNH GIÁ VÀ NHẬN XÉT



Để đánh giá ngưỡng tốt nhất trong FFT denoising, ta có thể dựa trên một số tiêu chí và phương pháp định lượng như sau:

- 1. Signal-to-Noise Ratio (SNR):** SNR là một thang đo quan trọng để đánh giá chất lượng của tín hiệu sau khi lọc nhiễu. Nó đo lường tỷ lệ giữa công suất tín hiệu mong muốn và công suất nhiễu. SNR cao hơn chỉ ra rằng tín hiệu có ít nhiễu hơn. Ta có thể tính SNR cho mỗi ngưỡng và so sánh kết quả để xác định ngưỡng nào mang lại chất lượng tín hiệu tốt nhất.
- 2. Root Mean Square Error (RMSE):** là một thước đo sai số bình phương trung bình giữa tín hiệu đã lọc và tín hiệu gốc (nếu có dữ liệu gốc). Ngưỡng nào mang lại RMSE thấp nhất có thể được coi là tối ưu nhất trong việc bảo toàn tín hiệu gốc mà vẫn loại bỏ nhiễu hiệu quả.
- 3. Visual Inspection:** Kiểm tra trực quan vẫn là một phương pháp quan trọng để có thể chọn ra đâu là tín hiệu tốt nhất. Ưu tiên chính vẫn là mục đích sử dụng để giữ lại những phần quan trọng nhất của tín hiệu.

RMSE

RMSE đo lường khoảng cách, trung bình, giữa các giá trị dự đoán và các giá trị thực tế, qua đó cung cấp một ước lượng về lỗi dự đoán của mô hình. Giá trị RMSE càng thấp, chất lượng dự đoán của mô hình càng tốt.

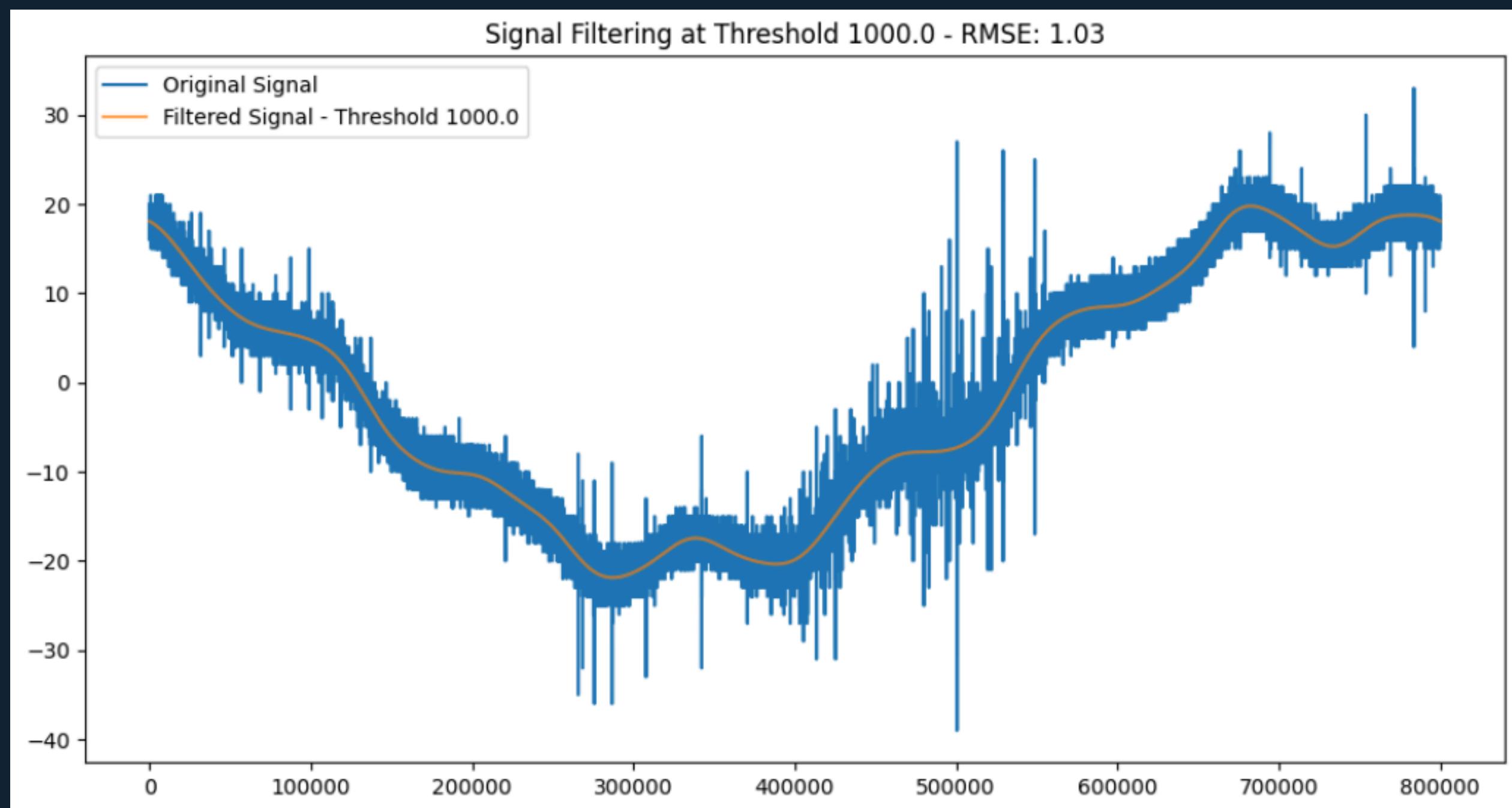
Công thức của RMSE:

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2}$$

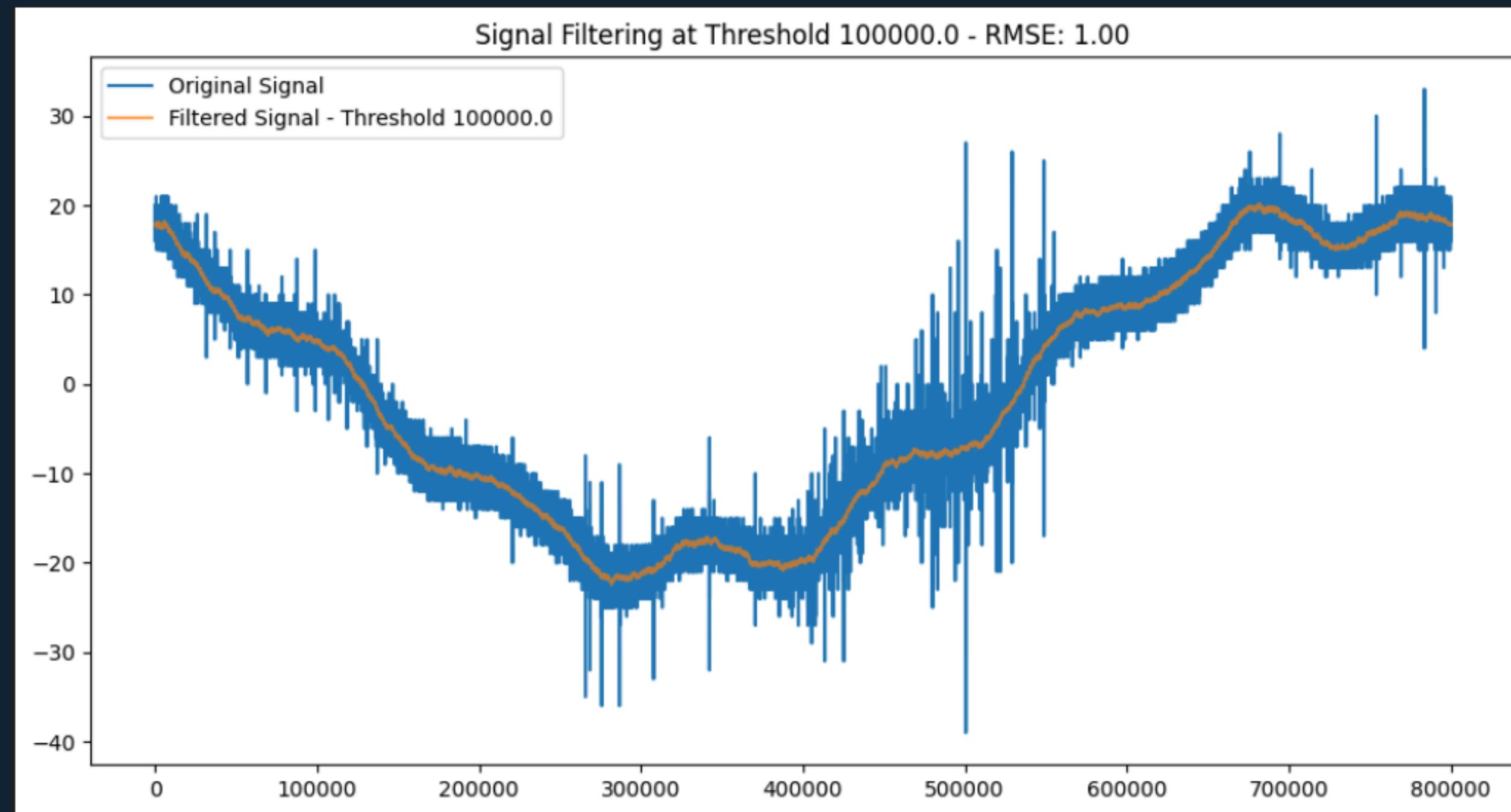
trong đó:

- N là số lượng quan sát.
- y_i là giá trị thực tế tại điểm dữ liệu thứ i .
- \hat{y}_i là giá trị dự đoán tại điểm dữ liệu thứ i .

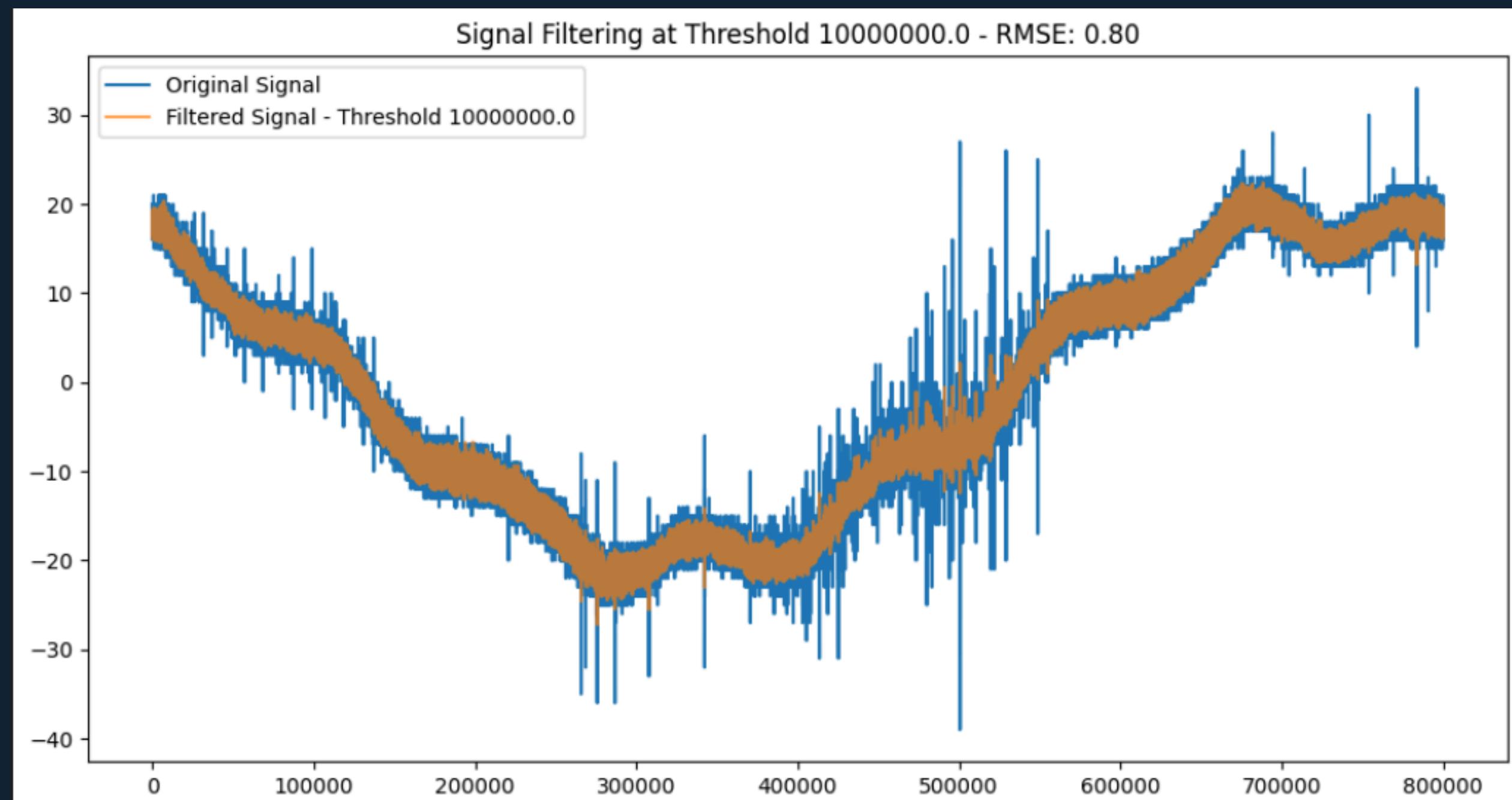
RMSE (1e3)



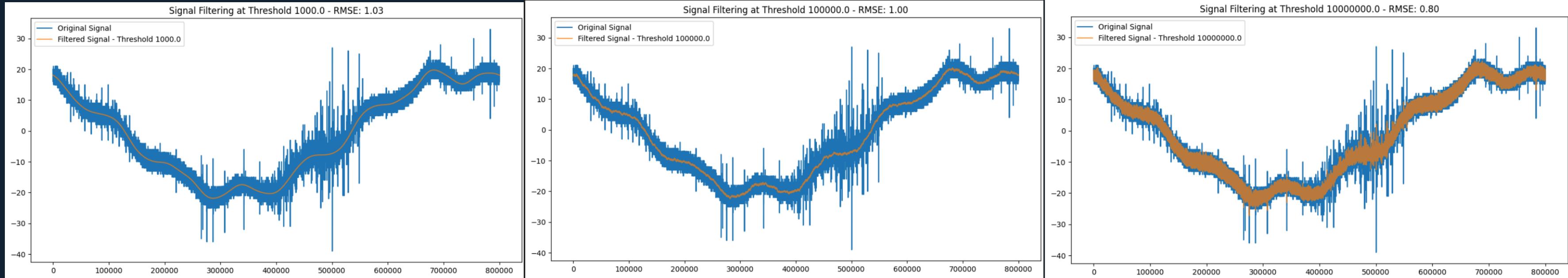
RMSE (1e5)



RMSE (1e7)



RMSE



Threshold 1000.0: RMSE = 1.03

Threshold 100000.0: RMSE = 1.00

Threshold 1000000.0: RMSE = 0.80

Đánh giá:

- Thresholds thấp (1000 và 100000): Những ngưỡng này không lọc bỏ đủ nhiễu trong tín hiệu, do đó tín hiệu sau khi lọc vẫn chứa nhiều thành phần nhiễu. Điều này dẫn đến giá trị RMSE cao hơn, cho thấy sự khác biệt đáng kể giữa tín hiệu gốc và tín hiệu đã lọc.
- Threshold 1000000.0: Ngưỡng này hiệu quả hơn trong việc loại bỏ nhiễu, làm giảm RMSE xuống còn 0.80. Điều này cho thấy tín hiệu đã lọc gần với tín hiệu gốc hơn so với các ngưỡng thấp hơn.

GNR

GNR: Giá trị SNR càng cao, chất lượng tín hiệu so với nhiễu càng tốt, cho thấy tín hiệu đã được lọc nhiễu hiệu quả hơn.

Công thức của SNR:

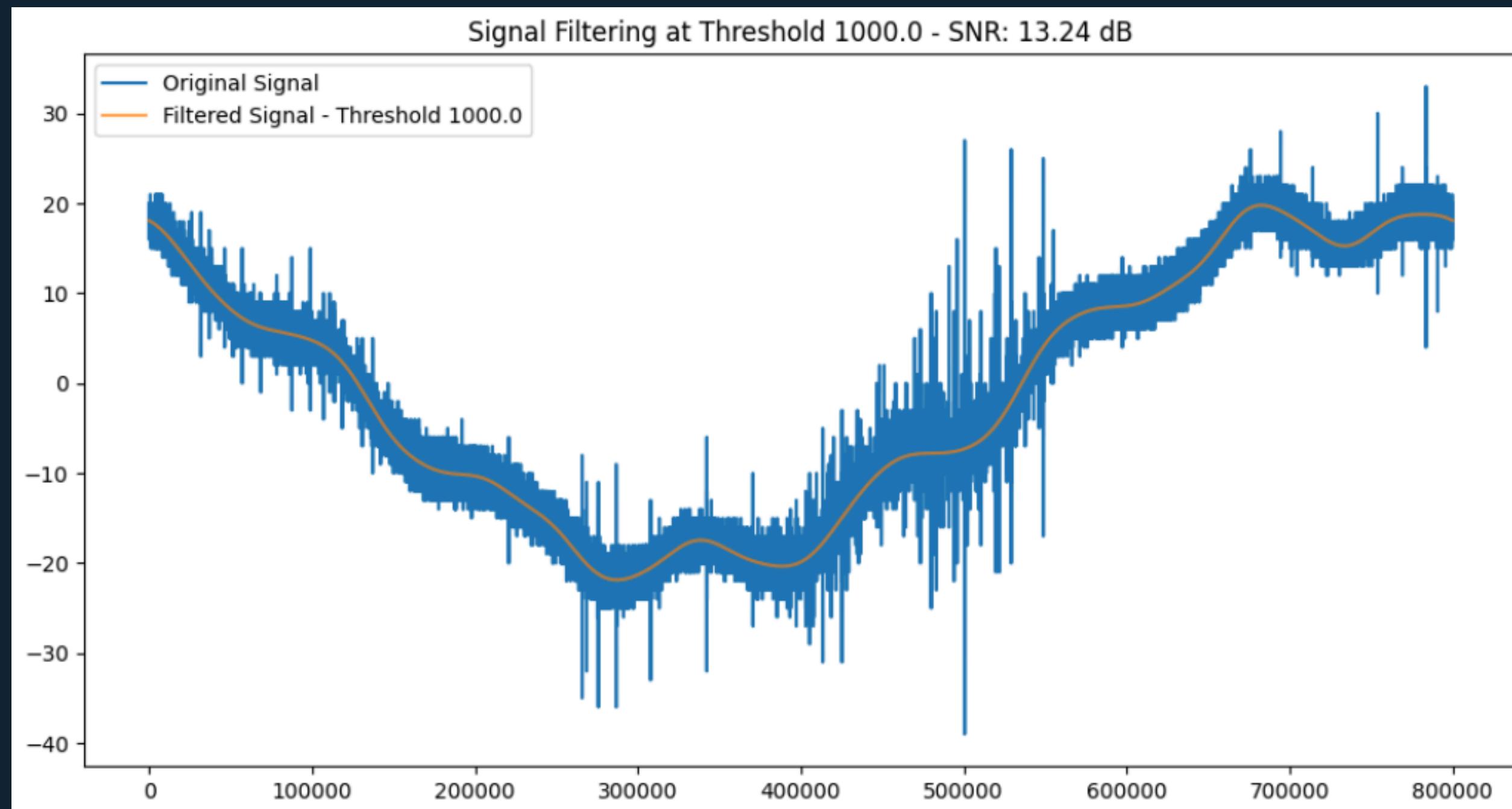
Signal-to-Noise Ratio (SNR) được tính bằng công thức:

$$\text{SNR (dB)} = 10 \cdot \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{noise}}} \right)$$

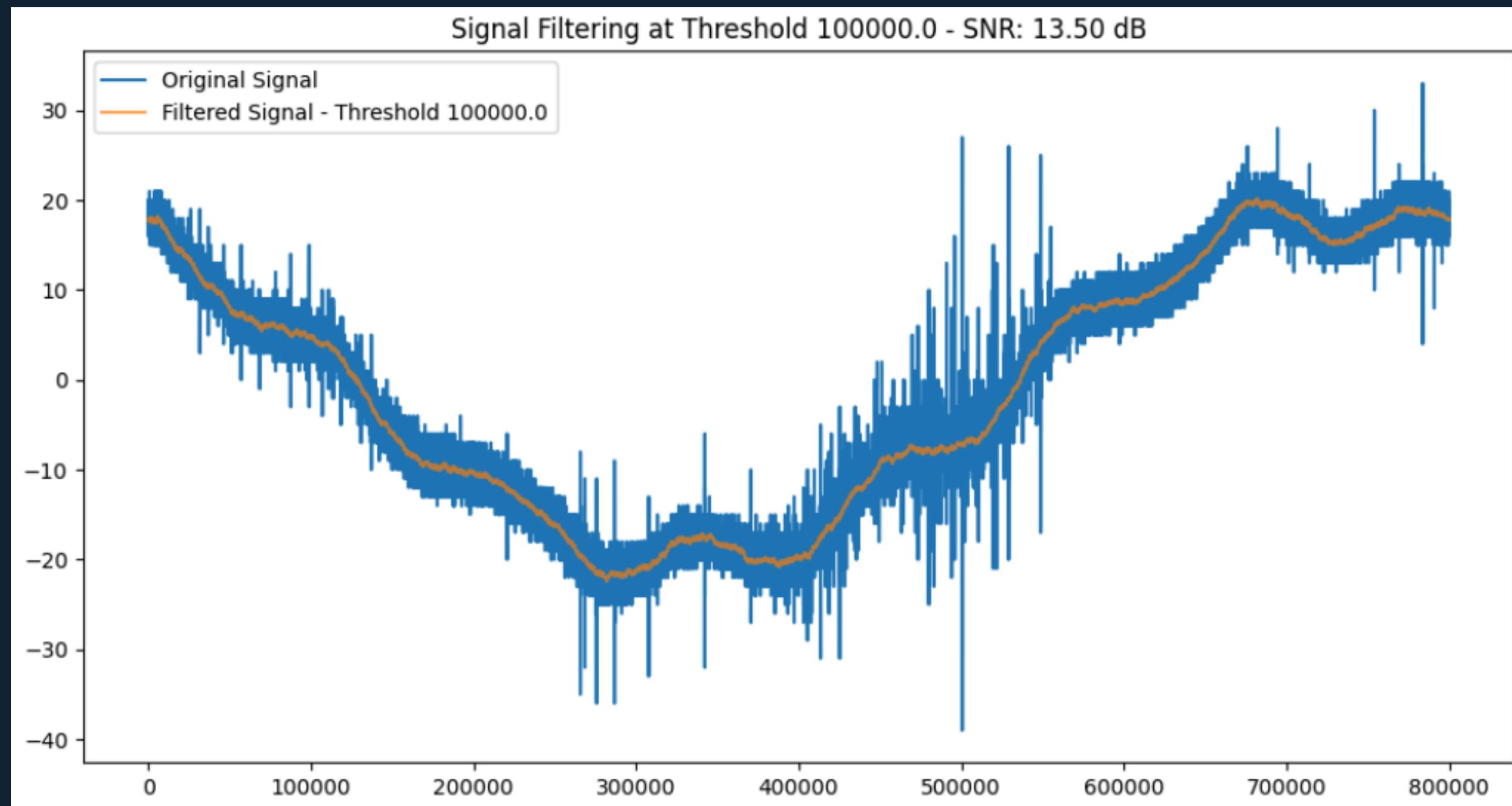
trong đó:

- P_{signal} là công suất trung bình của tín hiệu.
- P_{noise} là công suất trung bình của nhiễu, được tính bằng sự khác biệt giữa tín hiệu gốc và tín hiệu đã lọc.

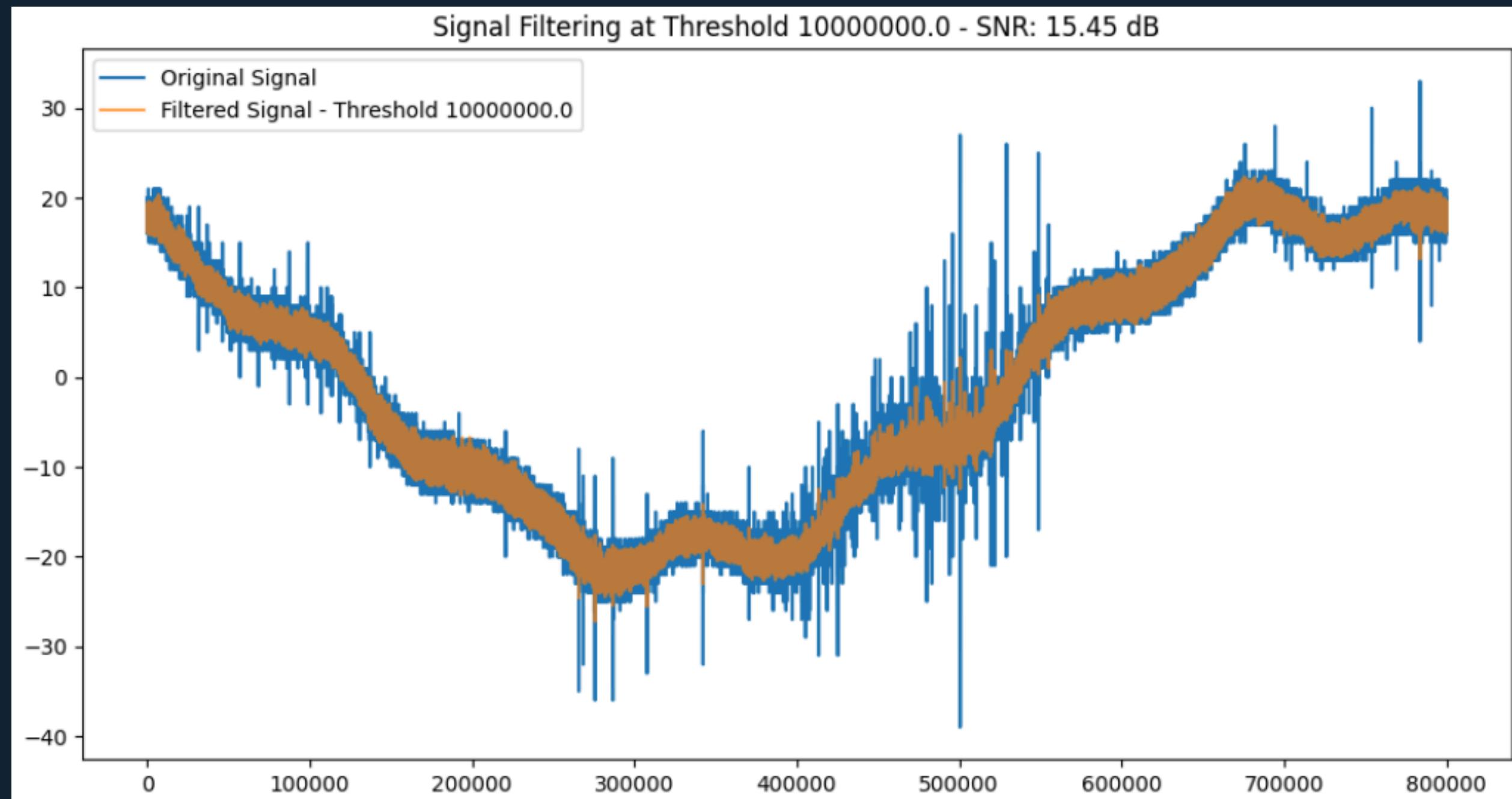
GNR (1e3)



GNR (1e5)



GNR (1e7)



1. Ngưỡng 1,000 ($1e3$): SNR = 13.24 dB

- cung cấp một mức độ giảm nhiễu vừa phải. SNR tương đối thấp, cho thấy mặc dù một số nhiễu được loại bỏ, nhưng vẫn còn khá nhiều nhiễu.

2. Ngưỡng 100,000 ($1e5$): SNR = 13.50 dB

- tăng nhẹ về SNR so với $1e3$ cho thấy việc khử nhiễu tốt hơn một chút. Nhưng sự cải thiện không đáng kể, cho thấy đặc tính nhiễu không khác biệt nhiều ở ngưỡng này.

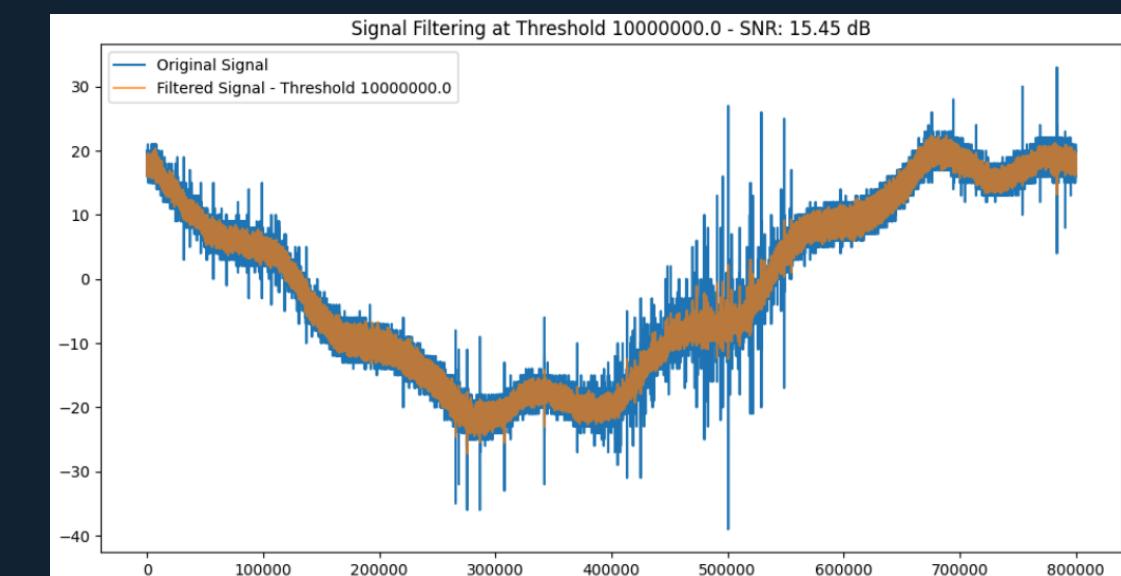
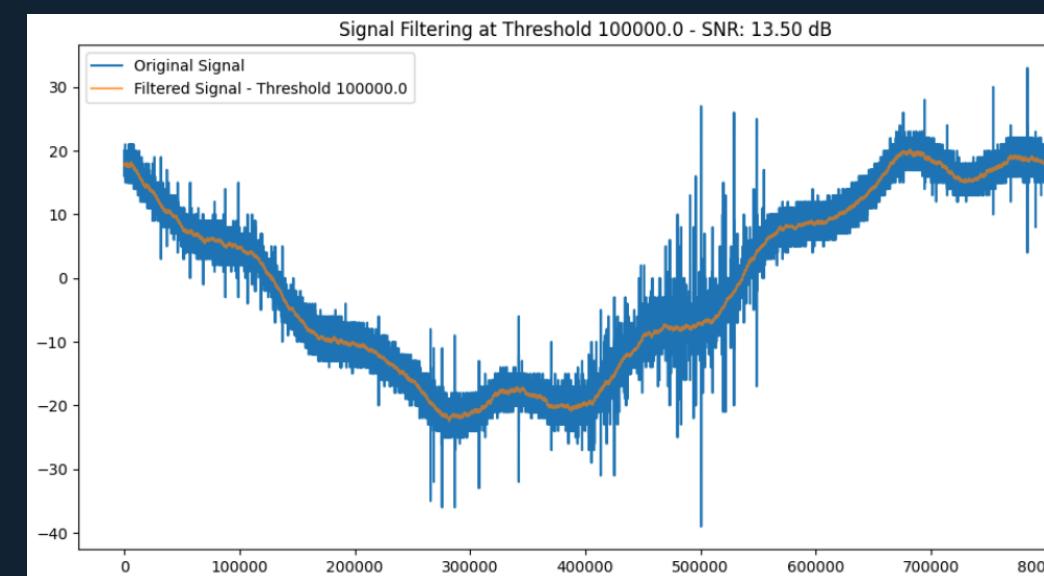
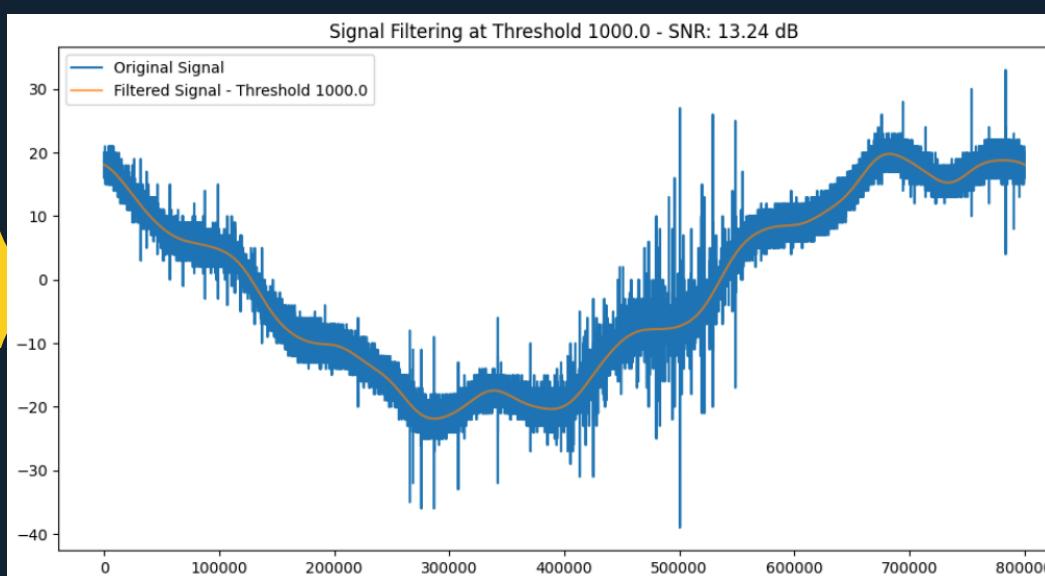
3. Ngưỡng 10,000,000 ($1e7$): SNR = 15.45 dB

- Ngưỡng này cung cấp một sự cân bằng tốt hơn giữa việc giảm nhiễu và bảo toàn tính toàn vẹn của tín hiệu. Cho thấy khả năng khử nhiễu hiệu quả hơn khi vẫn giữ lại các đặc điểm cần thiết của tín hiệu.

Xét về kết quả này, sự lựa chọn ngưỡng tối ưu phụ thuộc vào yêu cầu cụ thể của nhiệm vụ xử lý tín hiệu cho từng mục đích riêng:

- Nếu việc bảo toàn chi tiết tín hiệu càng nhiều càng tốt là rất quan trọng, thì ngưỡng khoảng $1e7$ có thể cung cấp sự cân bằng tốt nhất giữa việc giảm nhiễu và bảo toàn tín hiệu.

- Đối với đồ án này, việc trực quan xem xét tín hiệu bằng mắt thường thì ta chọn ngưỡng $1e5$ là ngưỡng tốt nhất để bảo toàn đặc tính của tín hiệu (khi được so với tín hiệu sau khi được làm mịn).





VI. Thử nghiệm và so sánh

VI. Hàm tuần tự.

1. Thư viện:

- **numpy**: Thư viện dùng để thực hiện các phép tính khoa học, chủ yếu là các phép tính trên mảng và ma trận.
- **numpy.fft**: Một module của NumPy cung cấp các chức năng để thực hiện phép biến đổi Fourier, bao gồm rfft (biến đổi Fourier nhanh cho mảng thực), irfft (biến đổi Fourier nghịch đảo cho mảng thực), và rfftfreq (tính tần số cho các thành phần trong FFT của một mảng thực).
- **pyarrow.parquet**: Thư viện dùng để đọc và viết dữ liệu dạng Parquet, một định dạng lưu trữ cột hiệu quả và nén.
- **time**: Thư viện để đo lường thời gian thực thi.

```
✓import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

✓def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
signals = np.array(signals_df).T.reshape(999//3, 3, 800000)

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:200, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")
```

Execution time for sequential processing of 200 signals: 11.13839 seconds

VI. Hàm tuần tự.

2. Hàm filtersignal:

- Hàm này nhận vào một **signal** (một mảng numpy biểu diễn tín hiệu) và một **threshold** (ngưỡng tần số).
- **rfft(signal)**: Áp dụng phép biến đổi Fourier nhanh cho tín hiệu, trả về biên độ của các thành phần tần số.
- **rfftfreq(signal.size, d=20e-3/signal.size)**: Tính tần số tương ứng cho mỗi thành phần trong FFT. **d** ở đây là khoảng cách giữa các mẫu trong tín hiệu.
- **fourier[frequencies > threshold] = 0**: Đặt các thành phần có tần số cao hơn ngưỡng về 0, nhằm lọc nhiễu hoặc các tần số không mong muốn.
- **irfft(fourier)**: Áp dụng biến đổi Fourier nghịch để chuyển các thành phần tần số đã được lọc trở lại dạng tín hiệu thời gian.

```
✓ import numpy as np
  from numpy.fft import rfft, irfft, rfftfreq
  import pyarrow.parquet as pq
  import time

✓ def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:200, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")
```

Execution time for sequential processing of 200 signals: 11.13839 seconds

VI. Hàm tuần tự.

3. Đọc dữ liệu:

- `parquetfilepath`: Đường dẫn tới file Parquet chứa dữ liệu.
- `pq.readtable(...).topandas()`: Đọc dữ liệu từ file Parquet và chuyển đổi sang DataFrame pandas.
- `np.array(signalsdf).T.reshape(999//3, 3, 800000)`: Chuyển đổi DataFrame thành một mảng numpy, xoay (transpose) và thay đổi hình dạng của mảng để phù hợp với dữ liệu gồm 333 tín hiệu, mỗi tín hiệu có 3 pha, và mỗi pha có 800,000 điểm dữ liệu.

```
✓import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

✓def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
signals = np.array(signals_df).T.reshape(999//3, 3, 800000)

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:200, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")
```

Execution time for sequential processing of 200 signals: 11.13839 seconds

VI. Hàm tuần tự.

4. Xử lý dữ liệu:

- `batch_signals = signals[:200, 0, :]`: Lấy 200 tín hiệu đầu tiên từ pha 0 của mỗi tín hiệu.

- `start_time = time.time()`: Ghi nhận thời điểm bắt đầu xử lý.

- `filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])`: Áp dụng hàm `filter_signal` cho mỗi tín hiệu trong batch, sử dụng list comprehension.

- `end_time = time.time()`: Ghi nhận thời điểm kết thúc xử lý.

- `print(...)`: In ra thời gian thực thi cho toàn bộ quá trình lọc tín hiệu.

```
✓import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

✓def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
signals = np.array(signals_df).T.reshape(999//3, 3, 800000)

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:200, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")
```

Execution time for sequential processing of 200 signals: 11.13839 seconds

VI. Song song FFT

Ý tưởng và cách thức hoạt động:

- Hàm bắt đầu bằng việc áp dụng biến đổi Fourier nhanh (rFFT) trên tín hiệu đầu vào. Điều này chuyển tín hiệu từ miền thời gian sang miền tần số, cho phép các phép tính tiếp theo được thực hiện trên các thành phần tần số của tín hiệu.
- Sau khi tín hiệu đã được chuyển đổi sang miền tần số, hàm tiếp tục tính toán các giá trị tần số tương ứng với mỗi phần tử trong mảng Fourier. Điều này được thực hiện để xác định xem thành phần tần số nào vượt qua ngưỡng lọc đã được đặt.
- Mảng Fourier và mảng tần số sau đó được chuyển từ bộ nhớ CPU sang bộ nhớ GPU. Điều này tận dụng khả năng xử lý song song của GPU để thực hiện các phép tính tiếp theo một cách hiệu quả hơn.
- Một kernel CUDA được triệu hồi để lọc các thành phần tần số. Trong hàm kernel, mỗi thread sẽ kiểm tra thành phần tần số tương ứng của nó có vượt quá ngưỡng đã định hay không. Nếu vượt quá, giá trị Fourier tương ứng sẽ được đặt thành 0, hiệu quả loại bỏ thành phần tần số đó khỏi tín hiệu.
- Sau khi lọc xong, mảng Fourier được chuyển trở lại từ GPU về CPU. Sau đó, một biến đổi Fourier nghịch (irFFT) được áp dụng để chuyển mảng Fourier đã lọc trở lại miền thời gian, tạo ra tín hiệu đã được lọc cuối cùng.

VI. Song song FFT

1. Thư viện:

- **numba.cuda**: Thư viện hỗ trợ việc viết mã Python chạy trên GPU thông qua CUDA.
- **numpy**: Thư viện hỗ trợ tính toán khoa học.
- **numpy.fft**: Chứa các hàm để thực hiện biến đổi Fourier.
- **pyarrow.parquet**: Thư viện dùng để đọc và viết dữ liệu theo định dạng Parquet.
- **time**: Dùng để đo lường thời gian thực thi của chương trình.

2. Hàm CUDA lọc tần số:

- **Hàm filter_frequencies_cuda** được dùng để lọc các thành phần tần số cao hơn ngưỡng chỉ định trong biểu đồ Fourier.
- **Sử dụng @cuda.jit** để biên dịch hàm này cho việc thực thi trên GPU.
- **Hàm nhận các tham số là fourier, frequencies và threshold.**
- **cuda.grid(1)** xác định vị trí của thread trong không gian một chiều.

```
✓ from numba import cuda
import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

@cuda.jit
def filter_frequencies_cuda(fourier, frequencies, threshold):
    idx = cuda.grid(1) # Lấy chỉ số của thread trong grid
    if idx < fourier.size:
        if frequencies[idx] > threshold:
            fourier[idx] = 0

def filter_signal_cuda(signal, threshold=1e8):
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)

    # Chuyển dữ liệu lên GPU
    d_fourier = cuda.to_device(fourier)
    d_frequencies = cuda.to_device(frequencies)

    # Gọi kernel
    threads_per_block = 256
    blocks_per_grid = (fourier.size + (threads_per_block - 1)) // threads_per_block
    filter_frequencies_cuda[blocks_per_grid, threads_per_block](d_fourier, d_frequencies, threshold)

    # Chuyển kết quả từ GPU về CPU và thực hiện IFFT trên CPU
    return irfft(d_fourier.copy_to_host())
```

VI. Song song FFT

3. Hàm xử lý tín hiệu dùng CUDA:

- **filter_signal_cuda** là hàm lọc tín hiệu sử dụng GPU. Hàm này thực hiện biến đổi Fourier nhanh (rfft) để chuyển tín hiệu từ miền thời gian sang miền tần số.
- Chuyển dữ liệu fourier và frequencies lên GPU để chuẩn bị cho xử lý.
- Tính toán số blocks và threads cần thiết cho việc phân phối công việc trên GPU.
- Gọi kernel đã biên dịch để lọc các tần số và sau đó chuyển kết quả trở lại CPU để thực hiện biến đổi Fourier nghịch (irfft).

4. Load dữ liệu và xử lý:

- Đọc dữ liệu tín hiệu từ file Parquet và chuẩn bị dữ liệu tín hiệu.
- Thực hiện "warm-up" cho GPU bằng cách gọi hàm lọc một lần.
- Đo thời gian thực thi khi xử lý batch tín hiệu đã chuẩn bị.

```
def filter_signal_cuda(signal, threshold=1e8):
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)

    # Chuyển dữ liệu lên GPU
    d_fourier = cuda.to_device(fourier)
    d_frequencies = cuda.to_device(frequencies)

    # Gọi kernel
    threads_per_block = 256
    blocks_per_grid = (fourier.size + (threads_per_block - 1)) // threads_per_block
    filter_frequencies_cuda[blocks_per_grid, threads_per_block](d_fourier, d_frequencies, threshold)

    # Chuyển kết quả từ GPU về CPU và thực hiện IFFT trên CPU
    return irfft(d_fourier.copy_to_host())

# Load dữ liệu tín hiệu
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
signals = np.array(signals_df).T.reshape((999//3, 3, 80000))
batch_signals = signals[:200, 0, :]

# Warm-up bằng cách gọi hàm một lần trước khi đo thời gian
warmup_signal = signals[0, 0, :]
filter_signal_cuda(warmup_signal, threshold=1e5)

# Áp dụng
start_time = time.time()
filtered_signals_cuda = np.array([filter_signal_cuda(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time with CUDA after warm-up: {end_time - start_time:.5f} seconds")
```

Execution time with CUDA after warm-up: 8.08892 seconds

So sánh giữa Tuần tự và Song song

Kết quả với 50 dữ liệu đầu

Tuần tự

```
[23] import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:50, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")
```

Execution time for sequential processing of 50 signals: 4.30065 seconds

Song song

```
def filter_frequencies_cuda(fourier, frequencies, threshold):
    idx = cuda.grid(1) # Lấy chỉ số của thread trong grid
    if idx < fourier.size:
        if frequencies[idx] > threshold:
            fourier[idx] = 0

def filter_signal_cuda(signal, threshold=1e8):
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)

    # Chuyển dữ liệu lên GPU
    d_fourier = cuda.to_device(fourier)
    d_frequencies = cuda.to_device(frequencies)

    # Gọi kernel
    threads_per_block = 256
    blocks_per_grid = (fourier.size + (threads_per_block - 1)) // threads_per_block
    filter_frequencies_cuda[blocks_per_grid, threads_per_block](d_fourier, d_frequencies)

    # Chuyển kết quả từ GPU về CPU và thực hiện IFFT trên CPU
    return irfft(d_fourier.copy_to_host())

# Load dữ liệu tín hiệu
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)].to_list())
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))
batch_signals = signals[:50, 0, :]

# Warm-up bằng cách gọi hàm một lần trước khi đo thời gian
warmup_signal = signals[0, 0, :]
filter_signal_cuda(warmup_signal, threshold=1e5)

# Áp dụng
start_time = time.time()
filtered_signals_cuda = np.array([filter_signal_cuda(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time with CUDA after warm-up: {end_time - start_time:.5f} seconds")
```

Execution time with CUDA after warm-up: 2.07690 seconds

So sánh giữa Tuần tự và Song song

Kết quả với 100 dữ liệu đầu

Tuần tự

```
[16] import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:100, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")

Execution time for sequential processing of 100 signals: 5.68985 seconds
```

Song song

```
iidx = cuda.grids[1] # Lấy chỉ số của thread trong grid
if idx < fourier.size:
    if frequencies[idx] > threshold:
        fourier[idx] = 0

def filter_signal_cuda(signal, threshold=1e8):
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)

    # Chuyển dữ liệu lên GPU
    d_fourier = cuda.to_device(fourier)
    d_frequencies = cuda.to_device(frequencies)

    # Gọi kernel
    threads_per_block = 256
    blocks_per_grid = (fourier.size + (threads_per_block - 1)) // threads_per_block
    filter_frequencies_cuda[blocks_per_grid, threads_per_block](d_fourier, d_frequencies, idx, threshold)

    # Chuyển kết quả từ GPU về CPU và thực hiện IFFT trên CPU
    return irfft(d_fourier.copy_to_host())

# Load dữ liệu tín hiệu
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))
batch_signals = signals[:100, 0, :]

# Warm-up bằng cách gọi hàm một lần trước khi đo thời gian
warmup_signal = signals[0, 0, :]
filter_signal_cuda(warmup_signal, threshold=1e5)

# Áp dụng
start_time = time.time()
filtered_signals_cuda = np.array([filter_signal_cuda(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time with CUDA after warm-up: {end_time - start_time:.5f} seconds

Execution time with CUDA after warm-up: 3.84354 seconds
```

So sánh giữa Tuần tự và Song song

Kết quả với 150 dữ liệu đầu

Tuần tự

```
[13] import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:150, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")
```

Execution time for sequential processing of 150 signals: 6.32776 seconds

Song song

```
def filter_frequencies_cuda(fourier, frequencies, threshold):
    idx = cuda.grid(1) # Lấy chỉ số của thread trong grid
    if idx < fourier.size:
        if frequencies[idx] > threshold:
            fourier[idx] = 0

def filter_signal_cuda(signal, threshold=1e8):
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)

    # Chuyển dữ liệu lên GPU
    d_fourier = cuda.to_device(fourier)
    d_frequencies = cuda.to_device(frequencies)

    # Gọi kernel
    threads_per_block = 256
    blocks_per_grid = (fourier.size + (threads_per_block - 1)) // threads_per_block
    filter_frequencies_cuda[blocks_per_grid, threads_per_block](d_fourier, d_frequencies)

    # Chuyển kết quả từ GPU về CPU và thực hiện IFFT trên CPU
    return irfft(d_fourier.copy_to_host())

# Load dữ liệu tín hiệu
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))
batch_signals = signals[:150, 0, :]

# Warm-up bằng cách gọi hàm một lần trước khi đo thời gian
warmup_signal = signals[0, 0, :]
filter_signal_cuda(warmup_signal, threshold=1e5)

# Áp dụng
start_time = time.time()
filtered_signals_cuda = np.array([filter_signal_cuda(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time with CUDA after warm-up: {end_time - start_time:.5f} seconds")
```

Execution time with CUDA after warm-up: 5.84256 seconds

So sánh giữa Tuần tự và Song song

Kết quả với 200 dữ liệu đầu

Tuần tự

```
import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:200, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time} seconds")
```

Execution time for sequential processing of 200 signals: 11.13839 seconds

Song song

```
# Chuyển kết quả từ GPU về CPU và thực hiện IFFT trên CPU
return irfft(d_fourier.copy_to_host())

# Load dữ liệu tín hiệu
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))
batch_signals = signals[:200, 0, :]

# Warm-up bằng cách gọi hàm một lần trước khi đo thời gian
warmup_signal = signals[0, 0, :]
filter_signal_cuda(warmup_signal, threshold=1e5)

# Áp dụng
start_time = time.time()
filtered_signals_cuda = np.array([filter_signal_cuda(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time with CUDA after warm-up: {end_time - start_time:.5f} seconds")
```

Execution time with CUDA after warm-up: 8.08892 seconds

So sánh giữa Tuần tự và Song song

Kết quả với 300 dữ liệu đầu

Tuần tự

```
import numpy as np
from numpy.fft import rfft, irfft, rfftfreq
import pyarrow.parquet as pq
import time

def filter_signal(signal, threshold=1e8):
    # Xử lý một tín hiệu duy nhất
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
    fourier[frequencies > threshold] = 0
    return irfft(fourier)

# Load data
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)])
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))

# Chọn 200 tín hiệu đầu tiên từ phase 0
batch_signals = signals[:300, 0, :]
start_time = time.time()
# Xử lý từng tín hiệu một trong batch sử dụng hàm filter_signal mới
filtered_batch_signals = np.array([filter_signal(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time for sequential processing of {len(batch_signals)} signals: {end_time - start_time:.5f} seconds")
```

Execution time for sequential processing of 300 signals: 15.80875 seconds

Song song

```
[ ] if frequencies[idx] > threshold:
        fourier[idx] = 0

def filter_signal_cuda(signal, threshold=1e8):
    fourier = rfft(signal)
    frequencies = rfftfreq(signal.size, d=20e-3/signal.size)

    # Chuyển dữ liệu lên GPU
    d_fourier = cuda.to_device(fourier)
    d_frequencies = cuda.to_device(frequencies)

    # Gọi kernel
    threads_per_block = 256
    blocks_per_grid = (fourier.size + (threads_per_block - 1)) // threads_per_block
    filter_frequencies_cuda[blocks_per_grid, threads_per_block](d_fourier, d_frequencies, threshold)

    # Chuyển kết quả từ GPU về CPU và thực hiện IFFT trên CPU
    return irfft(d_fourier.copy_to_host())

# Load dữ liệu tín hiệu
parquet_file_path = '/content/drive/MyDrive/LTSSUD/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
signals = np.array(signals_df).T.reshape((999//3, 3, 800000))
batch_signals = signals[:300, 0, :]

# Warm-up bằng cách gọi hàm một lần trước khi đo thời gian
warmup_signal = signals[0, 0, :]
filter_signal_cuda(warmup_signal, threshold=1e5)

# Áp dụng
start_time = time.time()
filtered_signals_cuda = np.array([filter_signal_cuda(signal, threshold=1e5) for signal in batch_signals])
end_time = time.time()
print(f"Execution time with CUDA after warm-up: {end_time - start_time:.5f} seconds")
```

Execution time with CUDA after warm-up: 13.02661 seconds

Số lượng tín hiệu	Phương thức	Tuần tự	Song song
50		4.30065 s	2.0769 s
100		5.68985 s	3.84354 s
150		6.32776 s	5.84256 s
200		11.13839 s	8.08892 s
300		15.80875 s	13.02661 s



Mục đích chính của việc áp dụng Fast Fourier Transform (FFT) để giảm nhiễu trong xử lý tín hiệu là để nâng cao chất lượng của tín hiệu bằng cách loại bỏ các thành phần tần số không mong muốn, thường là nhiễu. FFT là một công cụ mạnh mẽ cho phép chuyển đổi tín hiệu từ miền thời gian sang miền tần số, giúp dễ dàng phân tích và chỉnh sửa các thành phần tần số cụ thể của tín hiệu. Dưới đây là một số mục đích và lợi ích chính của việc sử dụng FFT để giảm nhiễu:

- **FFT cho phép xác định các thành phần tần số của tín hiệu, bao gồm cả tín hiệu mong muốn và nhiễu. Khi biết được thành phần tần số của nhiễu, có thể dễ dàng lọc hoặc giảm thiểu những tần số đó.**
- **Các thành phần tần số cao thường là nhiễu trong nhiều loại tín hiệu như tín hiệu âm thanh, tín hiệu hình ảnh, và tín hiệu điện từ các thiết bị điện tử hoặc cơ sở hạ tầng. Việc sử dụng FFT để loại bỏ các tần số này giúp làm giảm nhiễu và cải thiện độ rõ nét của tín hiệu.**
- **Tín hiệu sau khi được làm sạch từ nhiễu có độ chính xác cao hơn, làm cơ sở cho các phân tích tiếp theo và ra quyết định dựa trên dữ liệu. Điều này đặc biệt quan trọng trong các ứng dụng yêu cầu độ chính xác cao như y tế, tài chính, và an ninh.**
- **Trong một số trường hợp, việc áp dụng FFT cho phép phát hiện các sự kiện đặc biệt như phóng điện một phần trong các đường dây điện, dấu hiệu của các hiện tượng thiên nhiên trong dữ liệu thủy văn hoặc khí tượng, hoặc các dấu hiệu bệnh lý trong tín hiệu y tế.**
- **Bằng cách loại bỏ nhiễu và các thành phần tần số không cần thiết, FFT giúp giảm băng thông cần thiết cho việc truyền tải tín hiệu và giảm yêu cầu về không gian lưu trữ, qua đó tối ưu hóa hiệu quả của các hệ thống xử lý và truyền dữ liệu.**

V.4. Vấn đề gấp phái của nhóm:

Hiệu quả của song song hóa FFT chưa cho thấy hiệu suất đáng kể khi thời gian thực thi của hàm FFT ban đầu nhanh hơn so với hàm FFT đã song song hóa, nhóm vẫn đang nghiên cứu thêm và sau đây là những lý do có thể khiến việc này xảy ra:

1. Overhead của việc khởi tạo và song song hóa:

- Chi phí khởi tạo Numba: Numba sử dụng JIT compilation, có nghĩa là khi chạy hàm được trang bị Numba lần đầu, nó phải trải qua quá trình biên dịch. Quá trình này có thể gây ra độ trễ đáng kể, đặc biệt nếu hàm chỉ đc gọi một lần hoặc trên các tập dữ liệu nhỏ, chi phí biên dịch có thể lớn hơn thời gian tiết kiệm được từ việc song song hóa.
- Chi phí quản lý thread: việc tạo và quản lý các thread hoặc tác vụ song song có thể tiêu tốn nhiều tài nguyên hơn so với việc chạy tuần tự, đặc biệt khi số lượng tác vụ không đủ lớn để bù đắp cho chi phí này.

```
fourier = rfft(signal)
frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
fourier[frequencies > threshold] = 0
filtered_signal = irfft(fourier)
end_time = time.time()
print("Execution time of sequential function: {:.5f} seconds".format(end_time - start_time))
return filtered_signal

# Hàm song song với Numba
@njit
def filter_frequencies(fourier, frequencies, threshold):
    for i in range(fourier.size):
        if frequencies[i] > threshold:
            fourier[i] = 0
    return fourier

def filter_signal_parallel(fourier, frequencies, threshold=1e8):
    start_time = time.time()
    filtered_fourier = filter_frequencies(fourier, frequencies, threshold)
    filtered_signal = irfft(filtered_fourier) # Gọi hàm irfft bên ngoài @njit
    end_time = time.time()
    print("Execution time of parallel function: {:.5f} seconds".format(end_time - start_time))
    return filtered_signal

# Load the signal data
parquet_file_path = '/content/drive/MyDrive/Colab Notebooks/train.parquet'
signals_df = pq.read_table(parquet_file_path, columns=[str(i) for i in range(999)]).to_pandas()
signals = np.array(signals_df).T.reshape((999//3, 3, 800000)) # Reshape to 3 phases

# Chọn một tín hiệu để phân tích
signal = signals[0, 0, :]

# Chạy hàm tuần tự
filtered_signal_seq = filter_signal(signal, threshold=1e3)

# Chạy hàm song song (bao gồm thời gian warm-up cho Numba)
fourier = rfft(signal)
frequencies = rfftfreq(signal.size, d=20e-3/signal.size)
filtered_signal_par = filter_signal_parallel(fourier, frequencies, threshold=1e3)
```

Execution time of sequential function: 0.03549 seconds
Execution time of parallel function: 0.14465 seconds

- 2. Không hiệu quả trong việc sử dụng bộ nhớ: các tác vụ song song có thể truy cập bộ nhớ một cách không hiệu quả, dẫn đến tình trạng xung đột bộ nhớ cache và tăng latency. Điều này có thể làm giảm hiệu quả của việc song song hóa, đặc biệt nếu các tác vụ phụ thuộc vào dữ liệu cần được truy cập nhanh chóng và thường xuyên.**
- 3. Kích thước data không đủ lớn: song song hóa có thể không mang lại lợi ích đáng kể nếu kích thước dữ liệu không đủ lớn để các nhân xử lý có thể hoạt động hiệu quả. Trong trường hợp dữ liệu nhỏ, overhead của việc quản lý song song có thể lớn hơn lợi ích thu được.**
- 4. Khả năng tương thích và tối ưu của Numba: mặc dù Numba là một công cụ mạnh mẽ để tăng tốc Python, nhưng không phải tất cả các thao tác đều có thể được tối ưu hóa một cách hiệu quả bằng Numba. Ở đây, nhóm nghiên cứu FFT có thể không được song song hoàn toàn thông qua Numba, giới hạn hiệu quả của việc tối ưu hóa tổng thể.**



THANK you
so much

