BE 521: Final Project Part 1

Spring 2025

Adapted by Kevin Xie

Updated by Zhongchuan Xu

32 Points

Objective: Predict finger movements from ECoG Recordings

Due: April 10th

Project Overview

This final project involves predicting finger flexion using intracranial EEG (ECoG) in three human subjects. The data and problem framing come from the 4th BCI Competition (Miller et al. 2008). For the details of the problem, experimental protocol, data, and evaluation, please see the original 4th BCI Competition documentation (included as separate document). The remainder of the current document details your deliverables for part 1 of the project.

Important Deadlines

- Final Project Part 1 (Canvas)
- Due: April 10th
- 32 Points
- Team Registration
- Due: April 10th
- 5 Points
- Team Responsibilities (Canvas)
- Due: April 14th
- 3 Point
- Checkpoint 1, r > 0.33
- Due: April 17th
- 20 Points
- Checkpoint 2, r > 0.45
- Due: April 24th
- 15 Points
- End of competition, submit algorithm (Canvas):
- Due: April 25th
- 15 Points
- Final Report
- Due: April 27th
- 60 Points
- Competition results (Final class session)
- On: April 30th

The grading is structured so that going the extra mile is definitely rewarded. We want you to show what you've learned this semester, and to have some fun!

Writing Your Code

To get started with the final project we have provided a a series of method stubs for you to fill out. Your job for part 1 of the final project is to build a prediction pipeline that takes in the ECoG and dataglove finger angle recordings (serving as the data and labels respectively), then uses machine learning methods to generate predicted finger angles from the ECoG signals. The functions you will develop in this assignment are as follows:

- `get_windowed_feats` This function will take in raw ECoG data, and use the 2 following helper functions to filter the data, calculate sliding-window features.
- `filter_data` This function will apply a filter to the raw data and return cleaned data
- `get_features` This function will take in a window of cleaned data and return a vector of features for that window
- `create_R_matrix` This function will take in a feature matrix and return a response matrix as an adaptation of the optimal linear decoder method.

Optimal Linear Decoder

You will use a modified version of the **optimal linear decoder** method as described in Warland et al., 1997. We will recapitulate the method in this section, but consult the paper for more details. Our ultimate goal is to predict the angle of each finger as it moves over time using data recorded from the ECoG channels.

The position data is captured for 300 seconds, which you will split up into M total time bins, and the number of ECoG channels, v, is 61, 46, and 64 for subject 1, 2, and 3 respectively.

The paradigm we adapt here tries to predict finger angle at a given time window using ECoG features calculated over the preceding N time windows, using the following steps:

First, p features will be calculated across all v ECoG channels over M total time windows to get a feature matrix of shape $\left(M, (v \times p)\right)$

Then, following the approach that Warland et al., 1997 takes, we will construct a row vector corresponding to each time bin, that contains features for all the ECoG channels over the preceding *N* time bins (in the paper, spike counts are their features and they index neurons instead of ECoG channels). Thus, there will be a good amount of redundancy between row vectors of adjacent time bins, but that is okay.

Let $r_t^{c,\phi}$ be the value of the feature in window $t \in \{1, 2, \ldots, M\}$, channel $c \in \{1, 2, \ldots, v\}$ and with feature $\phi \in \{1, 2, \ldots, p\}$. Let the response matrix $R \in \mathrm{R}^{M \times (1+N \cdot p \cdot v)}$ be defined as:

$$R = \begin{bmatrix}
\mathbf{1} & r_1^{(1,1)} & r_1^{(1,1)} & \cdots & r_1^{(1,1)} & r_1^{(1,1)} & r_1^{(1,2)} & \cdots & r_1^{(1,2)} & \cdots & r_1^{(v,p)} & \cdots & r_1^{(v,p)} \\
\mathbf{1} & r_1^{(1,1)} & r_1^{(1,1)} & \cdots & r_1^{(1,1)} & r_2^{(1,1)} & r_1^{(1,2)} & \cdots & r_2^{(1,2)} & \cdots & r_1^{(v,p)} & \cdots & r_2^{(v,p)} \\
\mathbf{1} & r_1^{(1,1)} & r_1^{(1,1)} & \cdots & r_2^{(1,1)} & r_3^{(1,1)} & r_1^{(1,2)} & \cdots & r_3^{(1,2)} & \cdots & r_1^{(v,p)} & \cdots & r_3^{(v,p)} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
\mathbf{1} & r_1^{(1,1)} & r_2^{(1,1)} & \cdots & r_{N-1}^{(1,1)} & r_N^{(1,1)} & r_1^{(1,2)} & \cdots & r_N^{(1,2)} & \cdots & r_1^{(v,p)} & \cdots & r_N^{(v,p)} \\
\mathbf{1} & r_2^{(1,1)} & r_3^{(1,1)} & \cdots & r_N^{(1,1)} & r_{N+1}^{(1,1)} & r_2^{(1,2)} & \cdots & r_{N+1}^{(1,2)} & \cdots & r_2^{(v,p)} & \cdots & r_{N+1}^{(v,p)} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
\mathbf{1} & r_{M-N+1}^{(1,1)} & r_{M-N+2}^{(1,1)} & \cdots & r_{M-1}^{(1,1)} & r_M^{(1,1)} & r_{M-N+1}^{(1,2)} & \cdots & r_M^{(1,2)} & \cdots & r_{M-N+1}^{(v,p)} & \cdots & r_M^{(v,p)}
\end{bmatrix}$$

This is also referred to as the design or feature matrix, with each column being a predictor, or feature. The column of 1's accounts for the intercept term in linear regression/decoding. Here we are repeating the first windows $N-1$ times as the padding of the first windows. Make sure you understand what this matrix means before moving on.

We denote the target matrix as $Y \in \mathrm{R}^{M \times 5}$ and the prediction matrix (e.g. the predicted finger angles) as $\hat{Y} \in \mathrm{R}^{M \times 5}$. Note that in Warland et al., 1997, this quantity is referred to as the stimulus vector since they are talking about decoding the stimulus from neural data after it. We, on the other hand, are trying to decode finger positions using the ECoG data before it, but we can conveniently use the same method. Our goal is to find some optimal weight matrix or filter $f \in \mathrm{R}^{(1+N \cdot p \cdot v) \times 5}$ that minimizes the mean squared error:

$$f^* = \mathrm{argmin}_f \mathcal{L}(f) = \mathrm{argmin}_f \| Y - \hat{Y} \|^2,$$

where $\hat{Y} = Rf$

We start with the mean squared error (MSE) loss function:

$$\mathcal{L}(f) = \| Y - Rf \|^2 = (Y - Rf)^\top (Y - Rf)$$

To minimize the loss, we take the derivative with respect to the weight matrix f and set it equal to zero:

$$\frac{\partial \mathcal{L}}{\partial f} = -2 R^\top (Y - Rf) = 0$$

This implies:

$$R^\top Y = R^\top R f$$

Assuming that $R^\top R$ is invertible, we solve for f:

$$f = \left(R^\top R\right)^{-1} R^\top Y$$

This is the the analytic form for the optimal filter f that minimizes the MSE loss.

This equation should take a familiar form. Warland et al., 1997 don't refer to it as such, but this is exactly the same as linear regression, one of the most commonly used algorithms in practical machine learning. Not only is this algorithm remarkably powerful, but it has a

beautiful analytic form for learning the “weights” (here, the f matrix), a rarity in a field where almost all optimizations involve some sort of iterative algorithm. After learning the filter weights f, we can calculate the optimal predictions as:

$$\hat{Y} = Rf$$

Dataset

The dataset for part 1 is stored within `final_proj_part1_data.pkl`. The `.pkl` file type is a pickle file, which stores python objects. You can open the `.pkl` file with this code.

```
with open('final_proj_part1_data.pkl', 'rb') as f:
  proj_data = pickle.load(f)
```

This stores the data inside the file as a variable named proj_data.

NOTE: Python versions don't pickle with each other very well. This pickle file was made in Google Colab. If you are running your own installation of Python and cannot load the file, we recommend you use Colab

There are 3 subjects, each with their own Data Glove data (the glove they used to capture hand movements), and ECoG data. The data is represented as a dictionary with keys `'data_glove'` and `'ecog'`, storing the data glove and ecog data, respectively. These keys map to python lists of 3 items. Each item is an np.ndarray corresponding to a subject's data. See the pseudcode below.

```
proj_data = {
  'data_glove':[np.ndarray for subject 1, np.ndarray for subject 2, np.ndarray for subject 3],
  'ecog':[np.ndarray for subject 1, np.ndarray for subject 2, np.ndarray for subject 3]
}
```

All np.ndarray shapes for `'data_glove'` should be $(T, 5)$, where T is the number of samples in the signal, and 5 is the number of fingers.

The np.ndarray shapes for `'ecog'` are $(T, 61)$, $(T, 46)$, and $(T, 64)$, where T is the number of samples in the signal, and each subject had 61, 46, and 64 ecog channels, respectively.

The sampling rate of the data glove and ecog was 1000 Hz

Your task is to develop an algorithm to use the ECoG to predict finger movements that are captured by the Data Glove.

1. Getting Started (4 pts)

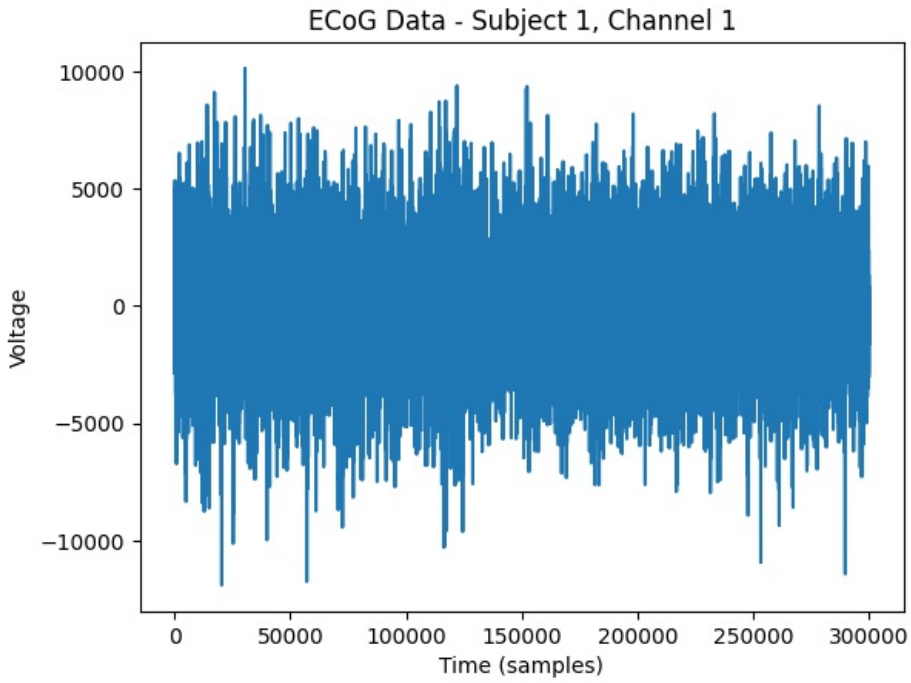
The following sections will walk you through the development of the prediction pipeline.

In [1]:
```
#Set up the notebook environment
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pickle
from scipy.stats import pearsonr
from scipy import signal as sig
```

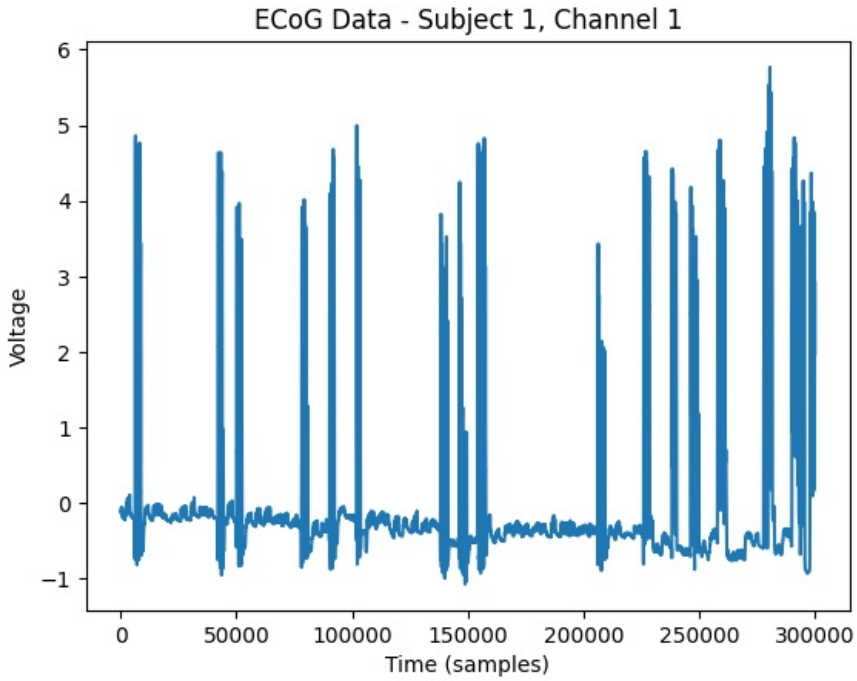
In [2]:
```
with open('final_proj_part1_data.pkl', 'rb') as f:
  proj_data = pickle.load(f)

ecog_data = proj_data['ecog']
dataglove_data = proj_data['data_glove']
```

Visualizing the data

In [3]:
```
import matplotlib.pyplot as plt

subject_index = 0  # Choose the subject (0, 1, or 2)
channel_index = 0  # Choose the channel to plot

plt.plot(ecog_data[subject_index][:, channel_index])
plt.xlabel('Time (samples)')
plt.ylabel('Voltage')
plt.title(f'ECoG Data - Subject {subject_index + 1}, Channel {channel_index + 1}')
plt.show()
```



In [4]:
```
import matplotlib.pyplot as plt

subject_index = 0  # Choose the subject (0, 1, or 2)
channel_index = 0  # Choose the channel to plot

plt.plot(dataglove_data[subject_index][:, channel_index])
plt.xlabel('Time (samples)')
plt.ylabel('Voltage')
plt.title(f'ECoG Data - Subject {subject_index + 1}, Channel {channel_index + 1}')
plt.show()
```



1.

Extract the dataglove and ECoG data for each subject from the pickle file. Feel free to copy the code snippet above. Split the data into a training and testing set (at least 50% of the data should be in the training set).

How many samples are there in the full ECoG recording (before splitting)? (1 pt)

How many samples do you have in your training set? In your testing set? (1 pt)

In [5]:
```
# We want a 70/30 train/test split
train_ratio = 0.7

# Loop through each subject
for subject_index in range(3):
    # Get the data for the current subject
```

```
    ecog_subject_data = ecog_data[subject_index]
    dataglove_subject_data = dataglove_data[subject_index]

    # Calculate the split index
    split_index = int(len(ecog_subject_data) * train_ratio)

    # Split the ECoG data
    ecog_train = ecog_subject_data[:split_index, :]
    ecog_test = ecog_subject_data[split_index:, :]

    # Split the Data Glove data
    dataglove_train = dataglove_subject_data[:split_index, :]
    dataglove_test = dataglove_subject_data[split_index:, :]

# no of samples before splitting
for subject_index in range(3):
    num_samples = len(ecog_data[subject_index])
    print(f"Subject {subject_index + 1}: {num_samples} samples")

# no of samples in the training set
# no of samples in the test set
for subject_index in range(3):
    num_train_samples = len(ecog_train)  # Assuming ecog_train is from the loop above
    num_test_samples = len(ecog_test)  # Assuming ecog_test is from the loop above
    print(f"Subject {subject_index + 1}: Training - {num_train_samples} samples, Testing - {num_test_samples} sa
```

```
Subject 1: 300000 samples
Subject 2: 300000 samples
Subject 3: 300000 samples
Subject 1: Training - 210000 samples, Testing - 90000 samples
Subject 2: Training - 210000 samples, Testing - 90000 samples
Subject 3: Training - 210000 samples, Testing - 90000 samples
```

Your answer here

2.

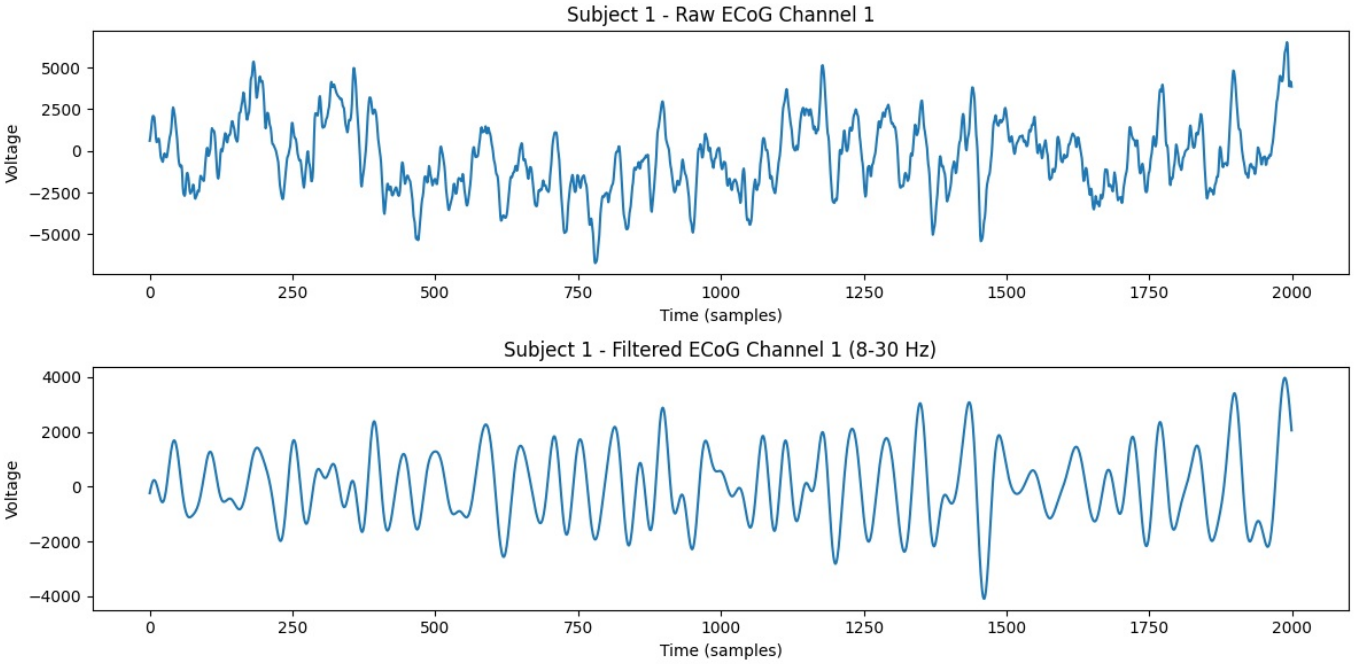
Next, complete the `filter_data` function. Test it using the raw data extracted in the prior step. What filter types and cutoff frequencies did you use? (2 pts)

- I used the bandpass filter with cut-off frequencies between 4 and 135 Hz as many papers mention it. I will try out the notch filter later for the competition.

In [6]:
```
from scipy import signal

# notch filter 60 hz , and add its harmonics
# for bandpass use 4 to 135

def filter_data(raw_data, fs, lowcut=4.0, highcut=135.0, order=4):
  """
  Write a filter function to clean underlying data.
  Filter type and parameters are up to you. Points will be awarded for reasonable filter type, parameters and ap
  Please note there are many acceptable answers, but make sure you aren't throwing out crucial data or adversly
  distorting the underlying data!

  Input:
    raw_eeg (samples x channels): the raw signal
    fs: the sampling rate (1000 for this dataset)
  Output:
    clean_data (samples x channels): the filtered signal
  """
  # Define filter parameters
  nyquist = fs / 2
  low = lowcut / nyquist
  high = highcut / nyquist
  b, a = sig.butter(order, [low, high], btype='band')
  filtered_data = sig.filtfilt(b, a, raw_data, axis=0)
  return filtered_data
```

In [7]:
```
# Extract raw data for Subject 1
raw_ecog = proj_data['ecog'][0]  # Shape: (T, 61)
fs = 1000  # Sampling frequency from document

# Apply filtering
filtered_ecog = filter_data(raw_ecog, fs, lowcut=8.0, highcut=30.0, order=4)

# Visualize raw vs filtered data for one channel
channel = 0
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
```

```
plt.plot(raw_ecog[:2000, channel])
plt.title(f'Subject 1 - Raw ECoG Channel {channel + 1}')
plt.xlabel('Time (samples)')
plt.ylabel('Voltage')

plt.subplot(2, 1, 2)
plt.plot(filtered_ecog[:2000, channel])
plt.title(f'Subject 1 - Filtered ECoG Channel {channel + 1} (8-30 Hz)')
plt.xlabel('Time (samples)')
plt.ylabel('Voltage')
plt.tight_layout()
plt.show()

# Verify shape preservation
print(f"Raw data shape: {raw_ecog.shape}")
print(f"Filtered data shape: {filtered_ecog.shape}")
```




```
Raw data shape: (300000, 61)
Filtered data shape: (300000, 61)
```

Your answer here

2. Calculating Features (12 points)

Here you will complete the `get_windowed_feats` and `get_features` functions.

1.

We will calculate features across sliding time windows. if we use a suggested window length of 100ms with a 50ms window overlap, how many feature windows, M, will we have if we computed features using all the data in a given subject? Feel free to re-use code from previous homeworks.

In [8]:
```
# 'fs' is the sampling rate (1000 Hz)

subject_index = 0  # Choose the subject (0, 1, or 2)
fs = 1000 #this was given

# Window parameters
window_length = 100  # in milliseconds
window_overlap = 50  # in milliseconds

# Convert window parameters to samples
window_length_samples = int(window_length * fs / 1000)
window_overlap_samples = int(window_overlap * fs / 1000)

# Calculate the number of windows
num_samples = ecog_data[subject_index].shape[0]  # Number of samples for the subject
num_windows = int((num_samples - window_length_samples) / (window_length_samples - window_overlap_samples)) + 1

# Print the number of windows (M)
print(f"Number of feature windows (M) for Subject {subject_index + 1}: {num_windows}")
```

```
Number of feature windows (M) for Subject 1: 5999
```

Your answer here

2.

Now complete the `get_features` function. Please create **4 or more** different features to calculate for each channel in each time window. Features may include the average time-domain voltage, or the average frequency-domain magnitude in consecutive 15Hz frequency bands, bandpower of relevant frequency bands, etc...

In [9]:
```
def get_features(filtered_window, fs=1000):
  """
    Write a function that calculates features for a given filtered window.
    Feel free to use features you have seen before in this class, features that
    have been used in the literature, or design your own!

    Input:
      filtered_window (window_samples x channels): the window of the filtered ecog signal
      fs: sampling rate
    Output:
      features (channels x num_features): the features calculated on each channel for the window
  """
  num_channels = filtered_window.shape[1]
  num_features = 6  # 1 for time-domain + 5 for frequency bands

  features = np.zeros((num_channels, num_features))

  for channel_index in range(num_channels):
      channel_data = filtered_window[:, channel_index]

      # Feature 1: Average time-domain voltage
      features[channel_index, 0] = np.mean(channel_data)

      # Calculate frequency-domain features using Welch's method
      frequencies, power_spectrum = signal.welch(channel_data, fs=fs, nperseg=len(channel_data))

      # Features 2,3,4: Average frequency-domain magnitude in specified bands
      freq_bands = [(5, 15), (20, 25), (75, 115)]
      for band_index, (low_freq, high_freq) in enumerate(freq_bands):
          freq_band_indices = np.where((frequencies >= low_freq) & (frequencies <= high_freq))
          features[channel_index, band_index + 1] = np.mean(power_spectrum[freq_band_indices])

      # features 5 and 6
      # line-length and energy
      line_length = np.sum(np.abs(np.diff(channel_data)))
      energy = np.sum(channel_data**2)

      features[channel_index, 4] = line_length
      features[channel_index, 5] = energy

  return features
```

3.

Now finish the `get_windowed_feats` function by putting the `filter_data` and `get_features` functions together to return a feature vector for each time window.

In [10]:
```
def get_windowed_feats(raw_ecog, fs, window_length, window_overlap):
  """
    Write a function which processes data through the steps of filtering and
    feature calculation and returns features. Points will be awarded for completing
    each step appropriately (note that if one of the functions you call within this script
    returns a bad output, you won't be double penalized). Note that you will need
    to run the filter_data and get_features functions within this function.

    Inputs:
      raw_eeg (samples x channels): the raw signal
      fs: the sampling rate (1000 for this dataset)
      window_length: the window's length
      window_overlap: the window's overlap
    Output:
      all_feats (num_windows x (channels x features)): the features for each channel for each time window
        note that this is a 2D array.
  """
# Filter the raw ECoG data
  filtered_ecog = filter_data(raw_ecog, fs)

  # Convert window parameters to samples
  window_length_samples = int(window_length * fs)  # e.g., 0.1 * 1000 = 100
  window_overlap_samples = int(window_overlap * fs)  # e.g., 0.05 * 1000 = 50
```

```
    step_size = window_length_samples - window_overlap_samples  # e.g., 100 - 50 = 50

    # Calculate the number of windows
    num_samples = filtered_ecog.shape[0]
    num_windows = (num_samples - window_length_samples) // step_size + 1
    if num_windows <= 0:
        raise ValueError("Not enough samples for at least one window. Adjust window_length or overlap.")

    # Initialize the features array
    num_channels = filtered_ecog.shape[1]
    num_features = 6  # From your get_features
    all_feats = np.zeros((num_windows, num_channels * num_features))

    # Extract features for each window
    for window_index in range(num_windows):
        start_sample = window_index * step_size  # Correct step size
        end_sample = start_sample + window_length_samples
        window_data = filtered_ecog[start_sample:end_sample, :]
        window_features = get_features(window_data, fs)
        all_feats[window_index, :] = window_features.reshape(1, -1)

    return all_feats
```

3. Creating the Response Matrix (6 points)

In this section, you will develop code for your *create_R_matrix* function

1.

For our set of 62 channels in subject 1, what would the dimensions of the R matrix be if we calculated 6 different feature types per channel, and $N = 3$ time bins where the number of total time bins M is the number you calculated in 2.1? (1pt)

The dimensions of the R matrix are given by:

R: (M, 1 + N * p * v)

Substituting the values:

R: (5999, 1 + 3 * 6 * 62)

R: (5999, 1117)

2.

We do not have feature data to fill out the first $N-1$ data rows in the R matrix that will be used to predict the first $N-1$ finger angles. One way to work around this is to append a copy of the first rows $N-1$ times of your feature matrix to the beginning of your feature matrix before calculating R. Make this adjustment in `create_R_matrix`, then compute the response matrix R. You can test whether your function is running correctly by running `create_R_matrix` with data from `testRfunction.pkl` using 3 windows and verifying that the quantity `np.mean(R)` is 25.4669 (5 points).

In [11]:
```
def create_R_matrix(features, N_wind):
    """
    Write a function to calculate the R matrix

    Input:
      features (num_windows x (channels x features)):
        the features you calculated using get_windowed_feats
      N_wind: number of windows to use in the R matrix

    Output:
      R (samples x (1+N_wind*channels*features))
    """

    num_windows = features.shape[0]
    num_features = features.shape[1]  # Total features per window (channels x features)

    # Initialize the R matrix
    R = np.zeros((num_windows, 1 + N_wind * num_features))

    # Add a column of 1s for the intercept term
    R[:, 0] = 1

    # Construct the rest of the R matrix
    for window_index in range(num_windows):
        # Get features from preceding N windows
```

```
        for wind_offset in range(N_wind):
            # Handle padding for the first N-1 windows
            prior_window_index = max(0, window_index - wind_offset)

            # Get features from prior_window_index
            start_col = 1 + wind_offset * num_features
            end_col = start_col + num_features
            R[window_index, start_col:end_col] = features[prior_window_index, :]

    return R
```

In [12]:
```
import pickle

# Load features from testRfunction.pkl
with open('testRfunction.pkl', 'rb') as f:
    test_features = pickle.load(f)

# Create R matrix
R_test = create_R_matrix(test_features, N_wind=3)

# Calculate and print the mean
print(np.mean(R_test))
```

```
25.46694219254388
```

Your answer here

4. ML Training and Testing (10 points)

Here we will use the optimal linear decoder framework to predict finger angles, and additionally you will use one or more models of your own choosing to make the prediction.

NOTE: This is a regression task - we are asking you to predict the movement of each finger for each subject.

1.

Calculate the linear filter f as defined above for all 5 finger angles using features calculated from your training data. You will have to first down-sample the finger flexion data so that your feature matrix, R, and your flexion data have the same number of time windows.

You will likely find np.matmul or operator @, and np.linalg.inv to be useful.

In [13]:
```
import numpy as np
N_wind = 3 #hyperparam

# 1. Extract features from training ECoG data
train_features = get_windowed_feats(ecog_train, fs, window_length, window_overlap)

# 2. Create the R matrix
R_train = create_R_matrix(train_features, N_wind)
print(R_train.shape)

# 3. Downsample the Data Glove data
# Calculate the downsampling factor based on window parameters
# downsampling_factor = int(window_length / (window_length - window_overlap))
num_windows_train = R_train.shape[0]
downsampling_factor = (len(dataglove_train) // num_windows_train)

# Downsample the Data Glove data using decimation
# dataglove_train_downsampled = dataglove_train[::downsampling_factor, :]
dataglove_train_downsampled = dataglove_train[::downsampling_factor, :][:num_windows_train, :]
print(dataglove_train_downsampled.shape)


# 4. Calculate the linear filter (f)
# Using the formula: f = (R^T * R)^-1 * R^T * Y
f = np.linalg.solve(R_train.T @ R_train, R_train.T @ dataglove_train_downsampled)
# Alternative: f = np.linalg.inv(R_train.T @ R_train) @ R_train.T @ dataglove_train_downsampled

print("Shape of linear filter (f):", f.shape)
```

```
(3, 1153)
(3, 5)
Shape of linear filter (f): (1153, 5)
```

Your answer here

2.

Try one other machine learning models using your features and finger angle labels. Look back through previous homeworks to get some ideas

In [14]:
```
from sklearn.linear_model import Ridge

# Assuming you have:
# - train_features: Extracted features from training ECoG data
# - dataglove_train_downsampled: Downsampled Data Glove data (target values)

# Create a Ridge regression model
ridge_model = Ridge(alpha=1.0)  # You can adjust the alpha parameter (regularization strength)

# Train the model
ridge_model.fit(train_features, dataglove_train_downsampled)

# Make predictions on the testing set (you'll need to extract features from testing data first)
# test_features = get_windowed_feats(ecog_test, fs, window_length, window_overlap)
# predictions = ridge_model.predict(test_features)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_ridge.py:254: UserWarning: Singular matrix in solv
ing dual problem. Using least-squares solution instead.
  warnings.warn(
```

Out[14]:

Ridge

Ridge()

Your answer here

3.

Produce predictions on the testing set for each finger angle.

Report your correlations here using the linear filter, and when using the other model(s) that you tried, as follows:

> For each subject, calculate the correlation coefficient between the predicted and test finger angles for each finger separately.

> You therefore should have 15 correlations: five per subject, with three subjects.

You will find pearsonr to be helpful and already imported.

In [15]:
```
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from scipy.stats import pearsonr

# Your parameters
N_wind = 5 # was 5 before, got 0.3 using ridge
fs = 1000
window_length = 0.1
window_overlap = 0.05
train_ratio = 0.7


# # Ensure get_windowed_feats matches the new get_features
# def get_windowed_feats(raw_ecog, fs, window_length, window_overlap):
#   # Filter the raw ECoG data
#   filtered_ecog = filter_data(raw_ecog, fs)

#   # Convert window parameters to samples
#   window_length_samples = int(window_length * fs)  # e.g., 0.1 * 1000 = 100
#   window_overlap_samples = int(window_overlap * fs)  # e.g., 0.05 * 1000 = 50
#   step_size = window_length_samples - window_overlap_samples  # e.g., 100 - 50 = 50

#   # Calculate the number of windows
#   num_samples = filtered_ecog.shape[0]
#   num_windows = (num_samples - window_length_samples) // step_size + 1
#   if num_windows <= 0:
#       raise ValueError("Not enough samples for at least one window. Adjust window_length or overlap.")

#   # Initialize the features array
#   num_channels = filtered_ecog.shape[1]
#   # Get the number of features from get_features output. This was 6 originally but updated to 5.
#   num_features = get_features(filtered_ecog[:window_length_samples,:], fs).shape[1]
#   all_feats = np.zeros((num_windows, num_channels * num_features))

#   # Extract features for each window
```

```
#   for window_index in range(num_windows):
#       start_sample = window_index * step_size  # Correct step size
#       end_sample = start_sample + window_length_samples
#       window_data = filtered_ecog[start_sample:end_sample, :]
#       window_features = get_features(window_data, fs)
#       all_feats[window_index, :] = window_features.reshape(1, -1)

#   return all_feats


# Initialize correlation dictionaries
correlations_linear = {0: [], 1: [], 2: []}
correlations_ridge = {0: [], 1: [], 2: []}

# Process each subject
for subject_idx in range(3):
    print(f"\nProcessing Subject {subject_idx + 1}")

    # Load and split data
    ecog_data = proj_data['ecog'][subject_idx]
    dataglove_data = proj_data['data_glove'][subject_idx]
    split_index = int(len(ecog_data) * train_ratio)

    ecog_train = ecog_data[:split_index, :]
    ecog_test = ecog_data[split_index:, :]
    dataglove_train = dataglove_data[:split_index, :]
    dataglove_test = dataglove_data[split_index:, :]

    # Extract features
    train_features = get_windowed_feats(ecog_train, fs, window_length, window_overlap)
    test_features = get_windowed_feats(ecog_test, fs, window_length, window_overlap)

    # Downsample dataglove data
    num_windows_train = train_features.shape[0]
    step_size = int((window_length - window_overlap) * fs)
    dataglove_train_downsampled = dataglove_train[::step_size][:num_windows_train, :]

    num_windows_test = test_features.shape[0]
    dataglove_test_downsampled = dataglove_test[::step_size][:num_windows_test, :]

    # Optimal Linear Decoder
    R_train = create_R_matrix(train_features, N_wind)
    f = np.linalg.solve(R_train.T @ R_train, R_train.T @ dataglove_train_downsampled)
    R_test = create_R_matrix(test_features, N_wind)
    y_pred_linear = R_test @ f

    # Calculate correlations for linear decoder
    for finger in range(5):
        corr, _ = pearsonr(dataglove_test_downsampled[:, finger], y_pred_linear[:, finger])
        correlations_linear[subject_idx].append(corr)

    # Ridge Regression with scaling
    scaler = StandardScaler()
    train_features_scaled = scaler.fit_transform(train_features)
    test_features_scaled = scaler.transform(test_features)

    ridge_model = Ridge(alpha=2.0)
    ridge_model.fit(train_features_scaled, dataglove_train_downsampled)
    y_pred_ridge = ridge_model.predict(test_features_scaled)

    # Calculate correlations for Ridge
    for finger in range(5):
        corr, _ = pearsonr(dataglove_test_downsampled[:, finger], y_pred_ridge[:, finger])
        correlations_ridge[subject_idx].append(corr)


# Report correlations
print("\n=== Optimal Linear Decoder Correlations ===")
for subject_idx in range(3):
    print(f"Subject {subject_idx + 1}: {[round(c, 4) for c in correlations_linear[subject_idx]]}")

print("\n=== Ridge Regression Correlations ===")
for subject_idx in range(3):
    print(f"Subject {subject_idx + 1}: {[round(c, 4) for c in correlations_ridge[subject_idx]]}")
```

```
Processing Subject 1

Processing Subject 2

Processing Subject 3

=== Optimal Linear Decoder Correlations ===
Subject 1: [np.float64(0.3446), np.float64(0.4618), np.float64(0.0761), np.float64(0.3035), np.float64(0.0732)]
Subject 2: [np.float64(0.3156), np.float64(0.1398), np.float64(0.1042), np.float64(0.2912), np.float64(0.0895)]
Subject 3: [np.float64(0.5024), np.float64(0.3525), np.float64(0.4482), np.float64(0.4497), np.float64(0.4071)]

=== Ridge Regression Correlations ===
Subject 1: [np.float64(0.2648), np.float64(0.3284), np.float64(0.0609), np.float64(0.2118), np.float64(0.0868)]
Subject 2: [np.float64(0.2352), np.float64(0.1513), np.float64(0.1047), np.float64(0.2181), np.float64(0.1038)]
Subject 3: [np.float64(0.4046), np.float64(0.2863), np.float64(0.3402), np.float64(0.4296), np.float64(0.3784)]
```

Interpolate the prediction using a cubic spline back to the original 1000 Hz sampling frequency. Constrain the first and last points to known values to avoid edge artifacts, applying zero- padding as needed to align with the original flexion trace

In [16]:

```python
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from scipy.stats import pearsonr
from scipy.interpolate import CubicSpline

# Your parameters
N_wind = 5
fs = 1000
window_length = 0.1
window_overlap = 0.05
train_ratio = 0.7

# Ensure get_windowed_feats matches the new get_features
# def get_windowed_feats(raw_ecog, fs, window_length, window_overlap):
#     filtered_ecog = filter_data(raw_ecog, fs)
#     window_length_samples = int(window_length * fs)
#     window_overlap_samples = int(window_overlap * fs)
#     step_size = window_length_samples - window_overlap_samples
#     num_samples = filtered_ecog.shape[0]
#     num_windows = (num_samples - window_length_samples) // step_size + 1
#     if num_windows <= 0:
#         raise ValueError("Not enough samples for at least one window.")
#     num_channels = filtered_ecog.shape[1]
#     num_features = get_features(filtered_ecog[:window_length_samples, :], fs).shape[1]
#     all_feats = np.zeros((num_windows, num_channels * num_features))
#     for window_index in range(num_windows):
#         start_sample = window_index * step_size
#         end_sample = start_sample + window_length_samples
#         window_data = filtered_ecog[start_sample:end_sample, :]
#         window_features = get_features(window_data, fs)
#         all_feats[window_index, :] = window_features.reshape(1, -1)
#     return all_feats

# Initialize correlation dictionaries
correlations_linear = {0: [], 1: [], 2: []}
correlations_ridge = {0: [], 1: [], 2: []}

# Process each subject
for subject_idx in range(3):
    print(f"\nProcessing Subject {subject_idx + 1}")

    # Load and split data
    ecog_data = proj_data['ecog'][subject_idx]
    dataglove_data = proj_data['data_glove'][subject_idx]
    split_index = int(len(ecog_data) * train_ratio)

    ecog_train = ecog_data[:split_index, :]
    ecog_test = ecog_data[split_index:, :]
    dataglove_train = dataglove_data[:split_index, :]
    dataglove_test = dataglove_data[split_index:, :]

    # Extract features
    train_features = get_windowed_feats(ecog_train, fs, window_length, window_overlap)
    test_features = get_windowed_feats(ecog_test, fs, window_length, window_overlap)

    # Downsample dataglove data for training
    num_windows_train = train_features.shape[0]
    step_size = int((window_length - window_overlap) * fs)
    dataglove_train_downsampled = dataglove_train[::step_size][:num_windows_train, :]

    num_windows_test = test_features.shape[0]
    dataglove_test_downsampled = dataglove_test[::step_size][:num_windows_test, :]
```

```
    # Optimal Linear Decoder
    R_train = create_R_matrix(train_features, N_wind)
    f = np.linalg.solve(R_train.T @ R_train, R_train.T @ dataglove_train_downsampled)
    R_test = create_R_matrix(test_features, N_wind)
    y_pred_linear = R_test @ f

    # Ridge Regression with scaling
    scaler = StandardScaler()
    train_features_scaled = scaler.fit_transform(train_features)
    test_features_scaled = scaler.transform(test_features)
    ridge_model = Ridge(alpha=1.0)
    ridge_model.fit(train_features_scaled, dataglove_train_downsampled)
    y_pred_ridge = ridge_model.predict(test_features_scaled)

    # Interpolate predictions to 1000 Hz
    # Time points for windowed predictions (center of each window)
    window_step = window_length - window_overlap  # 0.05 s
    t_windowed = np.arange(num_windows_test) * window_step + window_length / 2  # [0.05, 0.10, ...]

    # Full 1000 Hz time points for test set
    t_full = np.arange(len(dataglove_test)) / fs  # [0, 0.001, ..., 89.999]

    # Cubic spline interpolation with boundary constraints
    y_pred_linear_interp = np.zeros((len(dataglove_test), 5))
    y_pred_ridge_interp = np.zeros((len(dataglove_test), 5))

    for finger in range(5):
        # Linear Decoder interpolation
        cs_linear = CubicSpline(t_windowed, y_pred_linear[:, finger], bc_type=((1, 0.0), (1, 0.0)))
        y_pred_linear_interp[:, finger] = cs_linear(t_full)
        # Constrain first and last points to dataglove_test
        y_pred_linear_interp[0, finger] = dataglove_test[0, finger]
        y_pred_linear_interp[-1, finger] = dataglove_test[-1, finger]

        # Ridge interpolation
        cs_ridge = CubicSpline(t_windowed, y_pred_ridge[:, finger], bc_type=((1, 0.0), (1, 0.0)))
        y_pred_ridge_interp[:, finger] = cs_ridge(t_full)
        y_pred_ridge_interp[0, finger] = dataglove_test[0, finger]
        y_pred_ridge_interp[-1, finger] = dataglove_test[-1, finger]

    # Calculate correlations with full 1000 Hz dataglove_test
    for finger in range(5):
        corr_linear, _ = pearsonr(dataglove_test[:, finger], y_pred_linear_interp[:, finger])
        correlations_linear[subject_idx].append(corr_linear)

        corr_ridge, _ = pearsonr(dataglove_test[:, finger], y_pred_ridge_interp[:, finger])
        correlations_ridge[subject_idx].append(corr_ridge)

# Report correlations
print("\n=== Optimal Linear Decoder Correlations (Interpolated) ===")
for subject_idx in range(3):
    print(f"Subject {subject_idx + 1}: {[round(c, 4) for c in correlations_linear[subject_idx]]}")

print("\n=== Ridge Regression Correlations (Interpolated) ===")
for subject_idx in range(3):
    print(f"Subject {subject_idx + 1}: {[round(c, 4) for c in correlations_ridge[subject_idx]]}")
```

```
Processing Subject 1

Processing Subject 2

Processing Subject 3

=== Optimal Linear Decoder Correlations (Interpolated) ===
Subject 1: [np.float64(0.3644), np.float64(0.4484), np.float64(0.071), np.float64(0.2903), np.float64(0.0747)]
Subject 2: [np.float64(0.3176), np.float64(0.1308), np.float64(0.106), np.float64(0.2842), np.float64(0.085)]
Subject 3: [np.float64(0.5075), np.float64(0.3342), np.float64(0.4482), np.float64(0.453), np.float64(0.4101)]

=== Ridge Regression Correlations (Interpolated) ===
Subject 1: [np.float64(0.2994), np.float64(0.3658), np.float64(0.0648), np.float64(0.2335), np.float64(0.0768)]
Subject 2: [np.float64(0.2691), np.float64(0.1569), np.float64(0.0997), np.float64(0.2443), np.float64(0.1114)]
Subject 3: [np.float64(0.427), np.float64(0.312), np.float64(0.3765), np.float64(0.4384), np.float64(0.3882)]
```