Assignment 2

Mahdi Tabatabaei 400101515
Github Repository

Neuroscience of Learning, Memory, and Cognition

Dr. Aghajan

December 12, 2024

Contents

Introduction

In this document, you are required to implement functions for calculating brain connectivity and examine them on real data. The document consists of two parts. The first part involves the implementation of functions and analyzing their sensitivities, while the second part compares different metrics.

The foundation of brain connectivity studies is based on viewing the brain as a complex system. More specifically, in complex systems, we deal with a large number of components where these components interact in different ways, and part of the system's functionality is realized through these interactions. Similarly, the brain follows this paradigm. Different parts of the brain have their own functionalities, but to accomplish more complex tasks, interactions between various parts of the brain occur, functioning like a network of processors.

These interactions are generally divided into three categories:

1. **Structural Connectivity**: Structural connectivity is defined based on the anatomical structure of the brain, meaning that there is a direct neuronal connection between two parts of the brain.

2. **Functional Connectivity**: Functional connectivity does not necessarily indicate a direct anatomical connection. Instead, it is a mathematical representation of the synchronization of activity between two parts of the brain. For instance, two parts of the brain oscillating in a similar pattern at a specific frequency can be said to have functional connectivity. In mathematical terms, this connectivity implies *correlation*, not necessarily causation. This means it is unclear whether there is a sender and receiver or a shared source causing the synchronized oscillations in the two parts.

3. **Effective Connectivity**: In effective connectivity, the issue of directionality is partially addressed, but it still remains a mathematical description and cannot be directly and explicitly attributed to brain functionality. For example, this type of connectivity identifies whether the electrophysiological behavior of one specific region of the brain depends on the past behavior of another region. Thus, a quasi-causal relationship between these two regions can be assumed (i.e., one region sends information to another).

From a theoretical perspective, the last two types of brain connectivity (functional and effective) can be traced using EEG signals. This is due to the high temporal resolution of these signals. However, due to low spatial resolution, precise interpretation of these results poses challenges.

Implementation and Testing of Functions

Any real signal can be represented as an analytical signal. Using this approach, one can access the amplitude and instantaneous phase of a signal. If we denote the signal as $s(t)$, its analytical representation can be written as:

$$s_a(t) = s(t) + jH[s(t)]$$

where $H[s(t)]$ represents the Hilbert transform of the signal, defined as:

$$s(t) = \mathrm{Re}\{s_a(t)\}$$

The instantaneous phase and amplitude of the signal are defined as:

$$A_s(t) = |s_a(t)|$$

$$\phi_s(t) = \angle s_a(t)$$

In the subsequent exercises, we will deal with metrics derived from the phase and amplitude definitions mentioned above.

In the analysis of brain signals such as EEG signals, the relationships between the phase, amplitude, or combinations of the two across multiple electrodes or frequency bands can be used to define brain connectivity metrics. Broadly speaking, these can be categorized as:

- Phase-to-Phase Coupling
- Amplitude-to-Amplitude Coupling
- Phase-Amplitude Coupling

In this exercise, you are required to implement the three mentioned types of connectivity and test them. Additionally, one method for effective connectivity is introduced, covering the following metrics:

- Directed Phase Lag Index (dPLI) (Phase-to-Phase Coupling)
- Power-to-Power Correlation (Amplitude-to-Amplitude Coupling)
- Phase-Amplitude Coupling (PAC)

Directed Phase Lag Index

In brain connectivity studies, one of the most important tools for calculating connectivity is based on the synchronization of phase activity. This method calculates the directionality and synchronization of the phase signals between two signals, leading to the Phase Lag Index (PLI). It is one of the most widely used measures for assessing non-zero phase synchronization between signals. The mathematical definition of PLI is given as:

$$PLI(x, y) = |E[\text{sign}(\Delta\phi_{x,y})]|$$

This metric calculates the asymmetry of phase differences around zero. The idea is that if this measure is zero, it indicates that no consistent direction of information flow exists between two signals, and the phase difference is symmetrically distributed around zero. However, if the value deviates significantly from zero, it implies the presence of a predominant direction in the phase relationship. This is particularly useful in EEG signals, as it eliminates the effect of volume conduction, which does not represent true directional connectivity.

Another variation of this metric, which is used in this exercise, is the Directed Phase Lag Index (dPLI). The mathematical formula for dPLI is as follows:

$$dPLI = E[\text{Heaviside}(\Delta\phi_{x,y})]$$

Where:

$$\text{Heaviside}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0.5 & \text{for } x = 0 \\ 0 & \text{for } x < 0 \end{cases}$$

This metric measures the directional interaction between two signals by determining whether one signal systematically leads the other in phase. If the dPLI value is significantly greater than 0.5, it implies a consistent phase-leading direction between the two signals. Conversely, a

value close to 0.5 indicates no significant directional interaction, and the phase relationship is symmetric.

It is important to note that dPLI, like PLI, focuses on non-zero phase lag relationships, eliminating the impact of volume conduction. However, it is not capable of extracting causality but provides insights into directional phase synchronization.

$$PLI = 2|dPLI - 0.5|$$

Write a function that calculates the dPLI value between two signals of arbitrary (but equal) length. This function should take two one-dimensional arrays as input and output a single number between 0 and 1.

```
def heaviside(x):
    """Heaviside function H(x) as described:
    1 for x > 0, 0 for x < 0, 0.5 for x == 0
    """
    return np.where(x > 0, 1, np.where(x < 0, 0, 0.5))

def dpli(signal_x, signal_y):
    """Compute the dPLI between two signals, signal_x and signal_y."""

    signal_x = np.array(signal_x)
    signal_y = np.array(signal_y)

    phase_diff = np.angle(signal_x) - np.angle(signal_y)

    h = heaviside(phase_diff)

    dpli_value = np.mean(h)

    return dpli_value
```

In EEG signal processing, we face two major challenges. The first is noise, and the second is the effect of a common source. In the next section, we aim to evaluate the impact of noise and the common source after verifying the function's accuracy.

▬ *Delay Effect*

Generate three signals as follows:

$$x(t) = \sin(2\pi f_s t)$$

$$y(t) = \sin(2\pi f_s (t - t_0))$$

$$C(t) = \sin(2\pi f_c t)$$

Set f_s to 10 Hz and vary t_0 over the range $\left[0, \frac{1}{f_s}\right]$. Plot the dPLI values between signals x and y. Describe your observations.

- Consider a time range of 1 second with a time resolution of 500 Hz.
- Plot the graph for at least 100 time delay points.

```
    fs = 10
    fc = 4
    t_max = 1
```

```
    sampling_rate = 500
    t = np.linspace(0, t_max, int(sampling_rate * t_max), endpoint=False)

    def x(t):
        """Generate signal x(t) = sin(2 * pi * fs * t)"""
        return np.sin(2 * np.pi * fs * t)

    def y(t, t0):
        """Generate signal y(t) = sin(2 * pi * fs * (t - t0))"""
        return np.sin(2 * np.pi * fs * (t - t0))

    def C(t):
        """Generate signal C(t) = sin(2 * pi * fc * t)"""
        return np.sin(2 * np.pi * fc * t)

    t0_values = np.linspace(0, t_max, num=100)
    dpli_values = []

    for t0 in t0_values:
        signal_x = x(t)
        signal_y = y(t, t0)

        dpli_value = dpli(signal_x, signal_y)
        dpli_values.append(dpli_value)

    plt.figure(figsize=(8, 5))
    plt.plot(t0_values, dpli_values, label='dPLI', color='b')
    plt.title('dPLI between x(t) and y(t) for varying t0')
    plt.xlabel('Time shift (t0) [s]')
    plt.ylabel('dPLI')
    plt.grid(True)
    plt.legend()
    plt.show()
```

Observations

- The calculated dPLI (directed Phase Lag Index) remains constant for most values of t_0, indicating consistent phase relationship.
- A sudden increase in dPLI is observed near $t_0 = 1$, reflecting a shift in the phase synchronization between $x(t)$ and $y(t)$.
- The behavior suggests that the delay t_0 has minimal impact on phase synchronization for small values, but a significant effect as it approaches the sampling period $\frac{1}{f_s}$.

▬ *Common Source Effect*

To investigate the effect of a common source, set the delay t_0 to a constant value of $\frac{1}{500}$. This means that Y is a shifted version of X with a small time lag.

Generate the following signals (set f_c to 4 Hz):

$$x_c(t) = x(t) + \alpha C(t)$$

$$y_c(t) = y(t) + \beta C(t)$$

where α and β represent the influence of the common source.

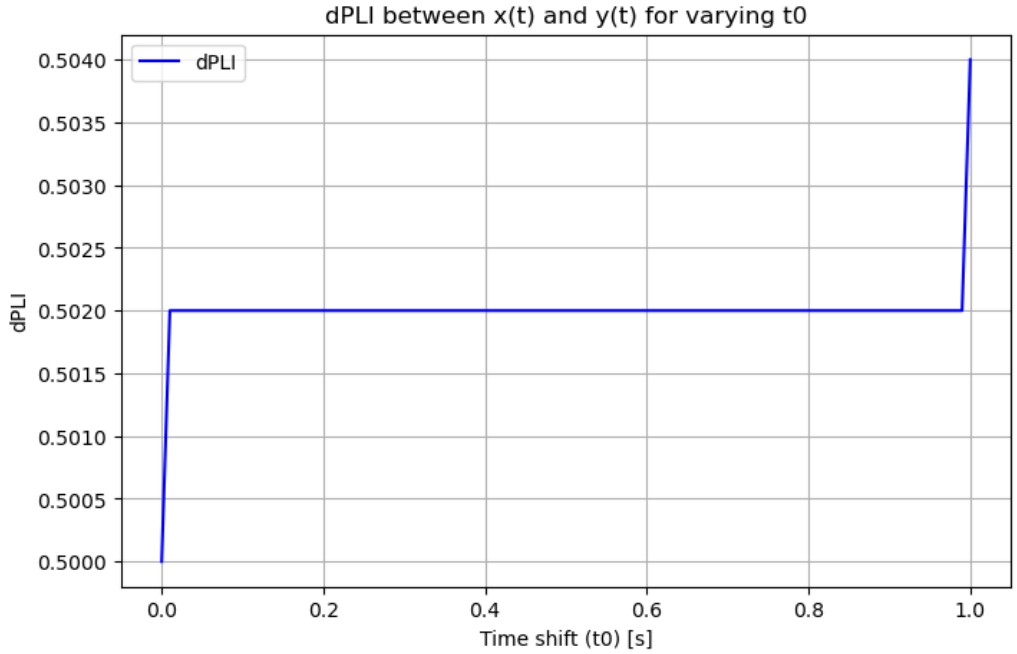


Figure 1: dPLI of x and y for 100 sample delay

- Calculate the dPLI values between x_c and y_c for different values of α and β and display the results as a heatmap using the `imshow` or `pcolormesh` commands. Describe your observations.
- Vary the coefficients α and β over a range of at least 50 points.

```
t0 = 0.002
alpha_values = np.linspace(0, 5, 50)
beta_values = np.linspace(0, 5, 50)

dpli_matrix = np.zeros((len(alpha_values), len(beta_values)))

for i, alpha in enumerate(alpha_values):
    for j, beta in enumerate(beta_values):
        signal_xc = x(t) + alpha * C(t)
        signal_yc = y(t, t0) + beta * C(t)

        dpli_matrix[i, j] = dpli(signal_xc, signal_yc)

plt.figure(figsize=(10, 6))
plt.pcolormesh(beta_values, alpha_values, dpli_matrix)
plt.colorbar(label='dPLI')
plt.title('Heatmap of dPLI between x_c(t) and yc(t) for varying alpha and beta')
plt.xlabel('Beta (Impact of common source on y(t))')
plt.ylabel('Alpha (Impact of common source on x(t))')
plt.grid(True)
plt.show()
```

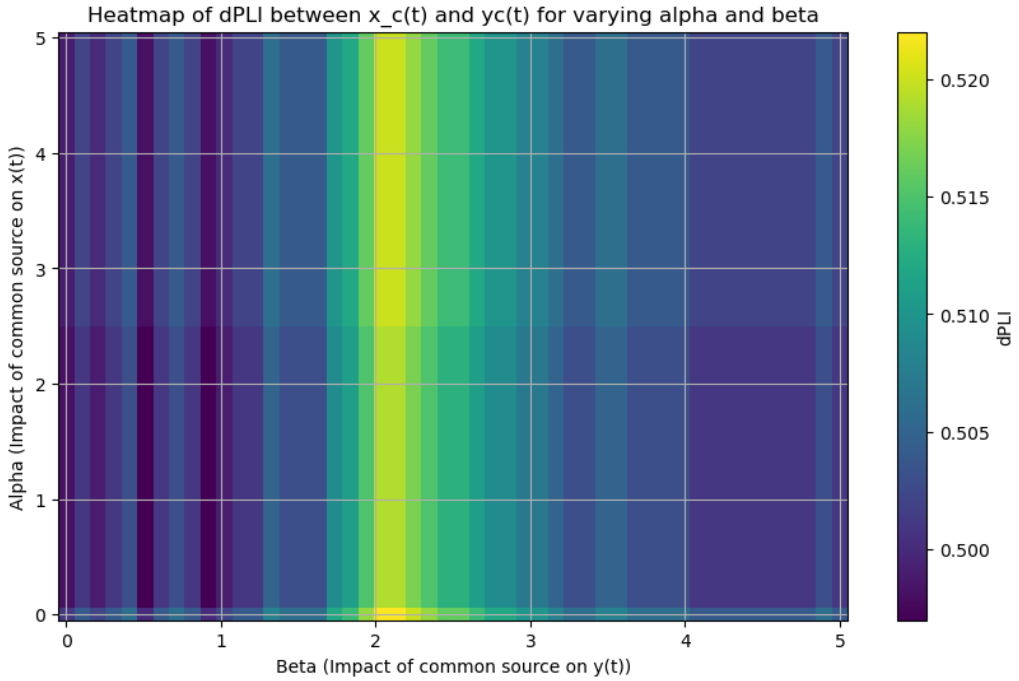


Figure 2: Common source effect on dPLI of 2 signals

Observations

- The heatmap illustrates the effect of the common source parameters α and β on the dPLI.
- dPLI is higher when α and β are closer to 2, indicating stronger phase synchronization due to the common source.
- The values of dPLI decrease as α or β deviate significantly from 2, suggesting reduced impact of the common source.
- The central bright region corresponds to maximal influence of the common source on phase synchronization.

▬ *Noise Effect*

To investigate the effect of noise, generate two independent signals with added noise and compare their dPLI values.

For this purpose, add noise to the signals x and y with different SNR values and calculate the dPLI values for the noisy signals. Display the results and describe your observations.

- Specifically, plot the results as a heatmap showing the dependency on SNR values ranging from -50 to 50 (for 100 different signal values).

```
snr_values = np.linspace(-50, 50, 100)  # SNR values from -50 dB to 50 dB
dpli_matrix = np.zeros((len(snr_values), len(snr_values)))

for i, snr_x in enumerate(snr_values):
    for j, snr_y in enumerate(snr_values):
        signal_x = x(t)
        signal_y = y(t, t0)
```

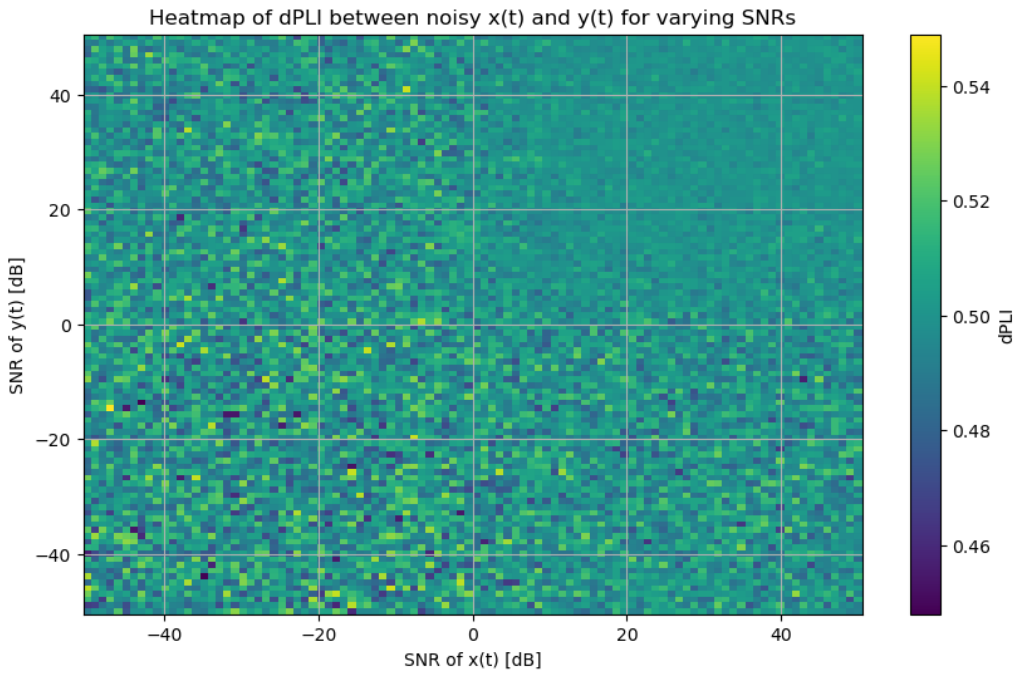


Figure 3: Noise effect on dPLI of 2 signals

```
            noisy_signal_x = add_noise(signal_x, snr_x)
            noisy_signal_y = add_noise(signal_y, snr_y)

            dpli_matrix[i, j] = dpli(noisy_signal_x, noisy_signal_y)

    plt.figure(figsize=(10, 6))
    plt.pcolormesh(snr_values, snr_values, dpli_matrix)
    plt.colorbar(label='dPLI')
    plt.title('Heatmap of dPLI between noisy x(t) and y(t) for varying SNRs')
    plt.xlabel('SNR of x(t) [dB]')
    plt.ylabel('SNR of y(t) [dB]')
    plt.grid(True)
    plt.show()
```

Observations

- The heatmap shows the effect of varying Signal-to-Noise Ratios (SNR) on the dPLI between noisy signals $x(t)$ and $y(t)$.
- dPLI values are relatively stable and close to 0.5 across a wide range of SNRs, indicating that noise minimally affects the phase relationship.
- No specific pattern or strong dependence on the SNR values of $x(t)$ or $y(t)$ is observed.
- The results suggest that the dPLI metric is robust to noise, maintaining phase synchronization even under high noise levels.

Power-to-Power Correlation

Another important metric for assessing the connectivity between two electrodes is domain correlation, which focuses on their level of synchrony. Synchrony in this context refers to the consistency and shared variations in the same domain. Various factors contribute to domain correlation, each of which plays a role in understanding signal relationships. Here, we examine correlations within the same domain (e.g., self-correlation) and across domains.

Correlation can be defined as:

$$\rho_{X,Y} = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}$$

where X and Y represent the domain signals of the two variables. The closer the correlation coefficient is to 1, the higher the synchrony between the signals, indicating that the two signals oscillate similarly and vary in the same direction. Conversely, a correlation coefficient closer to -1 implies strong synchrony in the opposite direction. A correlation coefficient closer to 0 indicates weaker or no dependency between the two signals.

To assess this metric, let us follow similar steps as in the previous sections to evaluate its performance.

Delay Effect

Generate three signals: $x(t)$, $y(t)$, and $C(t)$.
Examine the effect of time delay on this metric, similar to how it was done in the previous sections, and plot the corresponding graphs.

```
    t0_values = np.linspace(0, t_max, num=100)  # Time shifts from 0 to 1 second
    correlation_values = []

    for t0 in t0_values:
        signal_x = x(t)
        signal_y = y(t, t0)

        correlation_value = correlation(signal_x, signal_y)
        correlation_values.append(correlation_value)

    plt.figure(figsize=(8, 5))
    plt.plot(t0_values, correlation_values, label='Correlation', color='g')
    plt.title('Correlation between x(t) and y(t) for varying t0')
    plt.xlabel('Time shift (t0) [s]')
    plt.ylabel('Correlation')
    plt.grid(True)
    plt.legend()
    plt.show()

def correlation(signal_x, signal_y):
    """Compute the correlation between two signals, signal_x and signal_y."""

    covariance = np.cov(signal_x, signal_y)[0, 1]

    std_x = np.std(signal_x)
    std_y = np.std(signal_y)

    correlation_value = covariance / (std_x * std_y)

    return correlation_value
```

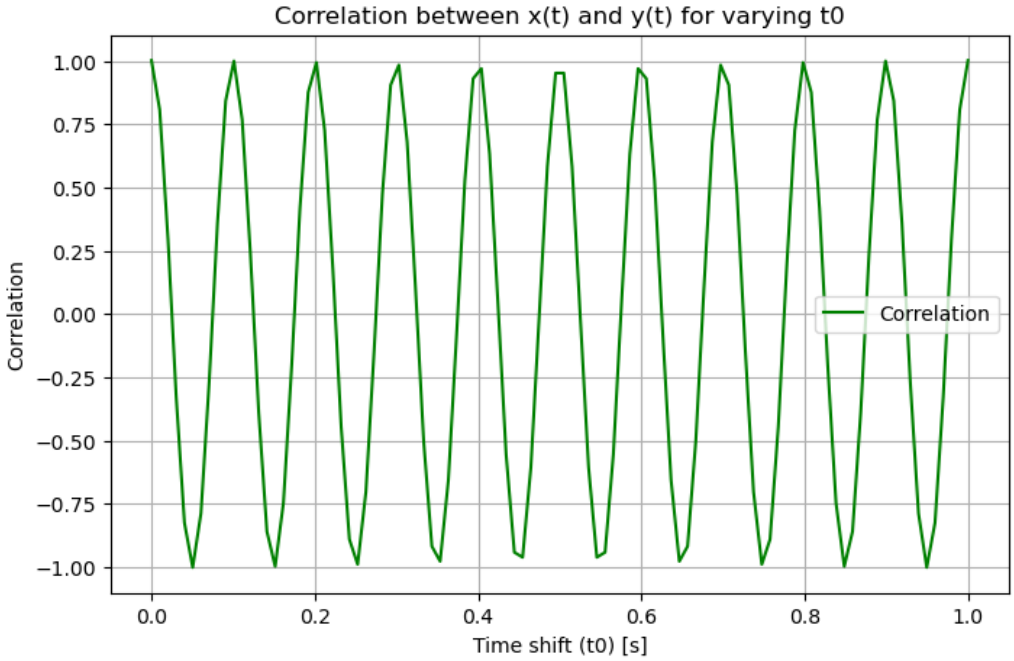


Figure 4: Correlation between x and y

▬ *Common Source Effect*

To investigate the effect of a common source:

First, generate two signals $n_1(t)$ and $n_2(t)$ as Gaussian white noise with zero mean and unit variance. Then, generate the following signals:

$$n_{1,c}(t) = n_1(t) + \alpha C(t)$$

$$n_{2,c}(t) = n_2(t) + \beta C(t)$$

Examine the connectivity between these signals for various values of α and β.

- Vary the coefficients over a range of at least 50 different values and display the results as a 3D heatmap.

```
    n1 = white_noise(len(t))
    n2 = white_noise(len(t))

    alpha_values = np.linspace(0, 5, 50)
    beta_values = np.linspace(0, 5, 50)

    correlation_matrix = np.zeros((len(alpha_values), len(beta_values)))

    for i, alpha in enumerate(alpha_values):
        for j, beta in enumerate(beta_values):
            n1_c = n1 + alpha * C(t)
            n2_c = n2 + beta * C(t)

            correlation_matrix[i, j] = correlation(n1_c, n2_c)

    plt.figure(figsize=(10, 6))
    plt.pcolormesh(beta_values, alpha_values, correlation_matrix, shading='
    auto')
```

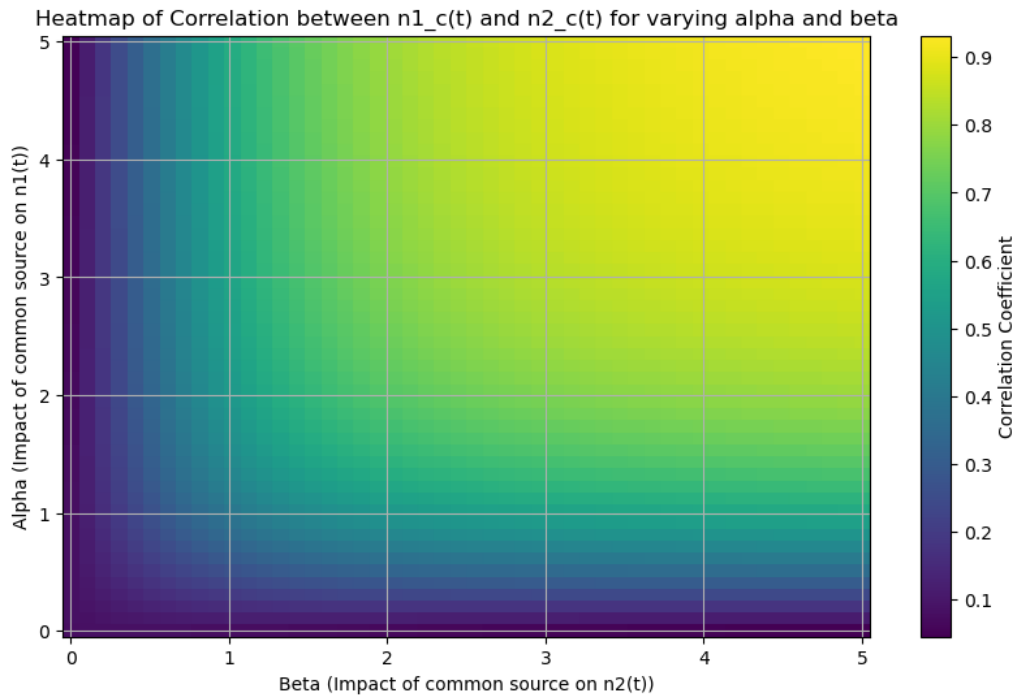


Figure 5: Common source effect on correlation of 2 noises

```
    plt.colorbar(label='Correlation Coefficient')
    plt.title('Heatmap of Correlation between n1_c(t) and n2_c(t) for varying alpha and beta')
    plt.xlabel('Beta (Impact of common source on n2(t))')
    plt.ylabel('Alpha (Impact of common source on n1(t))')
    plt.grid(True)
    plt.show()
```

Describe your observations.

Observations

- The heatmap shows the correlation between $n_{1,c}(t)$ and $n_{2,c}(t)$ as a function of α and β, which represent the impact of the common source on the signals.
- The correlation coefficient increases as both α and β increase, indicating a stronger influence of the common source on both signals.
- When α and β are close to zero, the correlation is minimal, showing that the signals are dominated by independent noise components.
- The gradient of the heatmap suggests that the common source contribution $(C(t))$ has a nonlinear impact on the correlation, particularly when α and β values are in the lower range.

Once again, assess the effect of the common source:

```
    t0 = 0.002

    alpha_values = np.linspace(0, 5, 50)
    beta_values = np.linspace(0, 5, 50)
```

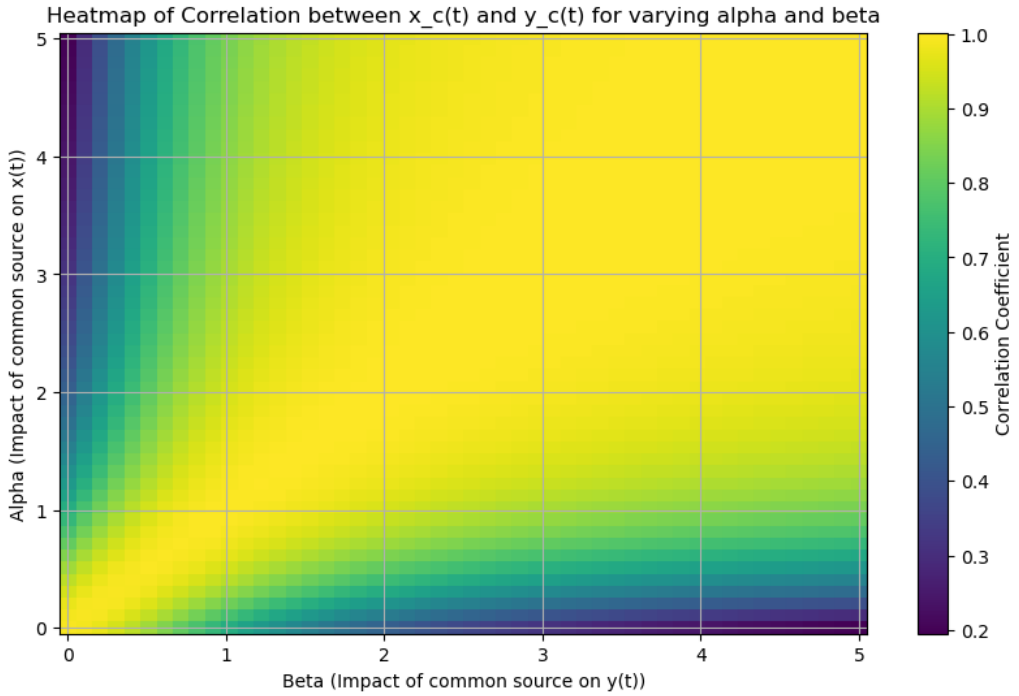


Figure 6: Common source effect on correlation of 2 signals

```
    dpli_matrix = np.zeros((len(alpha_values), len(beta_values)))

    for i, alpha in enumerate(alpha_values):
        for j, beta in enumerate(beta_values):
            signal_xc = x(t) + alpha * C(t)
            signal_yc = y(t, t0) + beta * C(t)

            dpli_matrix[i, j] = correlation(signal_xc, signal_yc)

    plt.figure(figsize=(10, 6))
    plt.pcolormesh(beta_values, alpha_values, dpli_matrix, shading='auto')
    plt.colorbar(label='Correlation Coefficient')
    plt.title('Heatmap of Correlation between x_c(t) and y_c(t) for varying
alpha and beta')
    plt.xlabel('Beta (Impact of common source on y(t))')
    plt.ylabel('Alpha (Impact of common source on x(t))')
    plt.grid(True)
    plt.show()
```

Based on the results from the previous section, determine whether it can be concluded that the common source introduces artificial connectivity.

Observations

- The heatmap in Figure 6 shows a clear dependency of the correlation coefficient on the parameters α and β, representing the common source's impact on $x_c(t)$ and $y_c(t)$.

- High values of α and β result in a correlation coefficient close to 1, indicating that the signals are almost entirely dominated by the common source.
- When comparing this result with previous findings, it is evident that the presence of a common source creates a strong phase and amplitude correlation between signals, reinforcing the influence of the common source.
- However, if α or β is minimal, the correlation approaches zero, suggesting little or no effect of the common source.
- Based on the comparison, the conclusion can be drawn that this method highlights the significance of the common source, but it might exaggerate non-realistic relationships when the parameters α and β are artificially set to high values. Thus, care must be taken in interpreting results for real-world scenarios.

▬ *Noise Effect*

Similarly, evaluate the effect of noise on this signal and state your observations.

```
snr_values = np.linspace(-50, 50, 100)
correlation_matrix = np.zeros((len(snr_values), len(snr_values)))

for i, snr_x in enumerate(snr_values):
    for j, snr_y in enumerate(snr_values):
        signal_x = x(t)
        signal_y = y(t,t0)

        noisy_signal_x = add_noise(signal_x, snr_x)
        noisy_signal_y = add_noise(signal_y, snr_y)

        correlation_matrix[i, j] = correlation(noisy_signal_x,
noisy_signal_y)

plt.figure(figsize=(10, 6))
plt.pcolormesh(snr_values, snr_values, correlation_matrix, shading='auto')
plt.colorbar(label='Correlation Coefficient')
plt.title('Heatmap of Correlation between noisy x(t) and y(t) for varying
SNRs')
plt.xlabel('SNR of x(t) [dB]')
plt.ylabel('SNR of y(t) [dB]')
plt.grid(True)
plt.show()
```

Observations

- The heatmap in Figure 7 shows the correlation coefficient between noisy signals $x(t)$ and $y(t)$ under varying Signal-to-Noise Ratios (SNRs).
- At high SNR values for both $x(t)$ and $y(t)$ (greater than 20 dB), the correlation coefficient approaches 1, indicating strong synchronization between the signals.
- For low SNR values (less than 0 dB), the correlation coefficient drops significantly, suggesting that noise dominates and weakens the relationship between the signals.

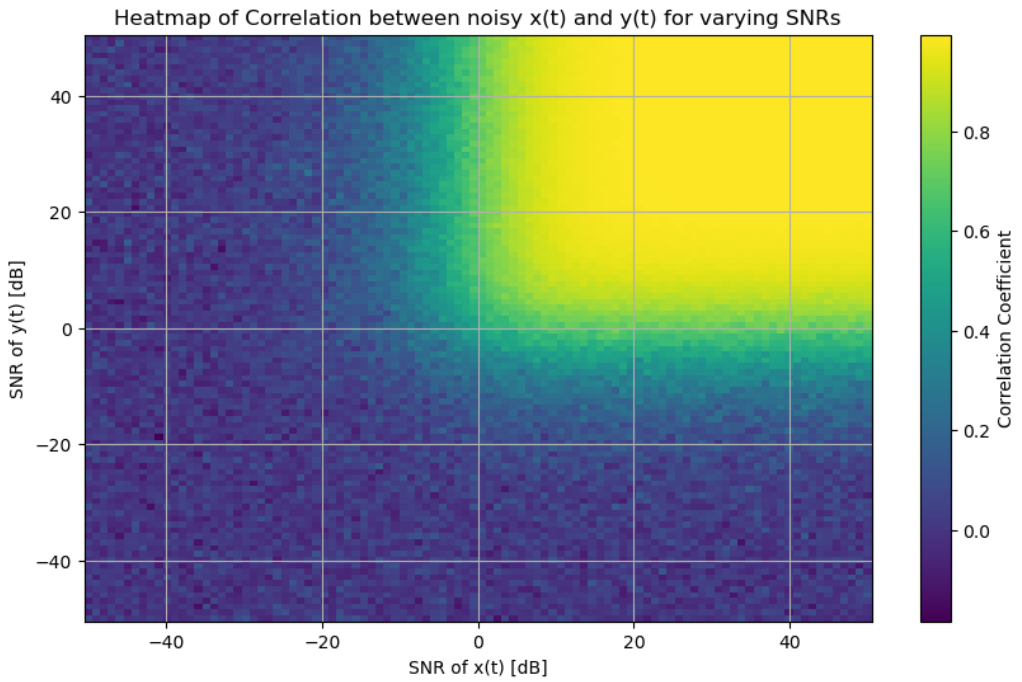


Figure 7: Noise effect on correlation of 2 signals

- The transition from low to high correlation is smooth, emphasizing that SNR thresholds significantly affect signal correlation strength.
- This result confirms that improving SNR enhances the detectability of the true correlation between noisy signals.

Phase-Amplitude Coupling

In this section, we examine and analyze one of the important methods for assessing connectivity in the phase and amplitude domain, namely, phase-amplitude coupling. Phase-amplitude coupling evaluates the extent of connectivity by examining the relationship between the phase of one frequency band and the amplitude of another frequency band. This method is particularly effective for analyzing cross-frequency coupling in different regions of the brain. For a more detailed explanation, you can refer to the discussion in this article.

In this exercise, you will implement a synthetic signal to calculate phase-amplitude coupling using two different methods. The goal is to:

First, create a time vector within the range $t = [0, 1]$ seconds with a sampling rate of 500 Hz.

Generate the signals $x_p(t)$ and $x_a(t)$ along with noise terms, defined as follows:

$$x_p(t) = k_p \cos(2\pi f_{\text{phase}} t) + \sigma_n n_1(t)$$

$$x_a(t) = k_a \left(\frac{(1-x)\cos(2\pi f_{\text{phase}} t) + x + 1}{2} \right) \cos(2\pi f_{\text{amp}} t) + \sigma_n n_2(t)$$

$n_1(t)$ and $n_2(t)$ are independent Gaussian white noise signals with zero mean and unit variance. The noise levels σ_n can be adjusted as needed.

Use the following parameter values:

$$k_p = 1,\ k_a = 1,\ f_{\text{phase}} = 5\,\text{Hz},\ f_{\text{amp}} = 60\,\text{Hz},\ x = 0,\ \sigma_n = 0$$

Plot the two signals on the same graph and describe their relationship.

```python
def generate_signals(kp, ka, fp, f_amp, chi, sigma):
    fs = 500
    t = np.linspace(0, 1, fs)

    noise1 = np.random.normal(0, 1, len(t)) * sigma
    noise2 = np.random.normal(0, 1, len(t)) * sigma

    xp = kp * np.cos(2 * np.pi * fp * t) + noise1
    xa = ka * ((1 - chi) * np.cos(2 * np.pi * fp * t) + (chi + 1) / 2) * np.cos(2 * np.pi * f_amp * t) + noise2

    return t, xp, xa
```

```python
    kp = 1
    ka = 1
    fp = 5
    f_amp = 60
    chi = 0
    sigma = 0

    fs = 500

    t, xp, xa = generate_signals(kp, ka, fp, f_amp, chi, sigma)

    plt.figure(figsize=(10, 6))
    plt.plot(t, xp, label='xp(t)')
    plt.plot(t, xa, label='xa(t)', linestyle='--')
    plt.title('Signal Comparison: xp(t) vs xa(t)')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Observations

- The plot in Figure 8 compares two signals: $x_p(t)$ and $x_a(t)$.
- $x_p(t)$ is a pure sinusoidal signal with a single frequency ($f_{phase} = 5$ Hz), while $x_a(t)$ combines two sinusoidal components with frequencies $f_{phase} = 5$ Hz and $f_{amp} = 60$ Hz.
- $x_a(t)$ exhibits amplitude modulation, which makes it more complex compared to $x_p(t)$.
- The periodicity of $x_p(t)$ matches f_{phase}, whereas $x_a(t)$ shows faster oscillations due to its higher frequency component (f_{amp}).
- Both signals are synchronized in time, but their waveforms differ significantly due to the added complexity in $x_a(t)$.

To calculate PAC between two signals, we need to extract the phase and amplitude of the signals in each frequency domain. For this purpose, we use time-frequency analysis based on

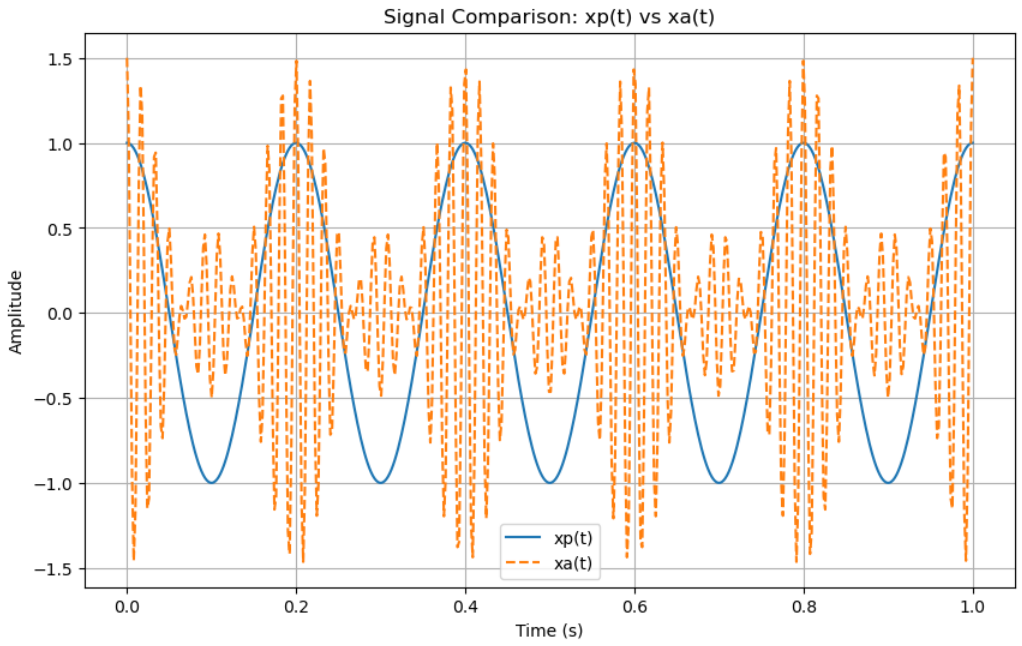


Figure 8: Plot of 2 signals

the wavelet transform. Further explanations regarding the wavelet transform and its implementation can be found in relevant sources. In this implementation, set the control parameter for the wavelet transform to `cmor1.5-0.5`.
Write a function that performs the following operations on the input signals $x_a(t)$ and $x_p(t)$:

- Perform time-frequency analysis on the two signals $x_a(t)$ and $x_p(t)$ to generate two complex matrices $X_a(t, f)$ and $X_p(t, f)$.
- Extract the phase matrix from $X_p(t, f)$ over the frequency range $f_p = [2, \ldots, 12]$ Hz, and name this matrix $\phi(t, f_p)$.
- Extract the amplitude matrix from $X_a(t, f)$ over the frequency range $f_a = [20, \ldots, 80]$ Hz, and name this matrix $A(t, f_a)$.
- Outputs of the function are the phase matrix $\phi(t, f_p)$ and the amplitude matrix $A(t, f_a)$.

```
def calculate_phase_amp(xa, xp, fs):
    """
    Processes two signals xa and xp using Continuous Wavelet Transform (CWT).

    Parameters:
    ----------
    xa : array-like
        Input signal A.
    xp : array-like
        Input signal P.
    fs : float
        Sampling frequency in Hz.

    Returns:
    -------
    phi_fp : 2D numpy.ndarray
        Phase of xp for frequencies 2-12 Hz. Shape: (11, time_points)
```

```
    A_fa : 2D numpy.ndarray
        Amplitude of xa for frequencies 20-80 Hz. Shape: (61, time_points)
    """

    f_p = np.arange(2, 13)   # 2 to 12 Hz (inclusive)
    f_a = np.arange(20, 81)  # 20 to 80 Hz (inclusive)

    frequencies = np.unique(np.concatenate((f_p, f_a)))

    frequencies = frequencies[(frequencies >= 2) & (frequencies <= 80)]

    wavelet = 'cmorl1.5-1.0'

    central_freq = pywt.central_frequency(wavelet)
    scales = central_freq * fs / frequencies

    Xa, _ = pywt.cwt(xa, scales, wavelet, sampling_period=1/fs)
    Xp, _ = pywt.cwt(xp, scales, wavelet, sampling_period=1/fs)

    phi = np.angle(Xp)  # Phase in radians

    idx_f_p = np.where((frequencies >= 2) & (frequencies <= 12))[0]
    idx_f_a = np.where((frequencies >= 20) & (frequencies <= 80))[0]

    phi_fp = phi[idx_f_p, :]

    A = np.abs(Xa)

    A_fa = A[idx_f_a, :]

    return phi_fp, A_fa
```

Now, using the time-frequency phase and amplitude matrices, calculate PAC using two different methods.

▬ *MLV Method*

To implement this method, for each frequency pair, calculate the instantaneous PAC value using the phase and amplitude domains. The output magnitude represents the PAC value, as shown below:

$$PAC(n,m) = \left| \frac{1}{N} \sum_{t=0}^{N-1} A(t, f_a(n)) e^{j\phi(t, f_p(m))} \right|$$

where n and m are indices corresponding to amplitude and phase frequencies, respectively.

Write a function that takes the phase matrix $\phi(t, f_p)$ and the amplitude matrix $A(t, f_a)$ as inputs and computes the PAC values for each pair of phase frequency f_p and amplitude frequency f_a using the MVL method.

```
def calculate_pac_MLV(phi, A):
    """
    Calculate the Phase-Amplitude Coupling (PAC) between phase and amplitude
    matrices using the MVL method,
    utilizing vectorized operations for efficiency.

    Parameters
    ----------
    phi : ndarray of shape (T, Fp)
        Phase matrix, where T is the number of time points and Fp is the
    number of phase frequencies.
```

```
        phi(t, fp) gives the phase at time t for phase frequency fp.
    A : ndarray of shape (T, Fa)
        Amplitude matrix, where T is the number of time points and Fa is the
    number of amplitude frequencies.
        A(t, fa) gives the amplitude at time t for amplitude frequency fa.

    Returns
    -------
    PAC : ndarray of shape (Fa, Fp)
        PAC matrix, where each element PAC[fa, fp] is the PAC value between
    the fa-th amplitude frequency
        and the fp-th phase frequency.
    """
    complex_matrix = A[:, :, np.newaxis] * np.exp(1j * phi[:, np.newaxis, :])

    PAC = np.abs(np.mean(complex_matrix, axis=0))

    return PAC
```

▬ *MI Method*

Another method for assessing phase-amplitude coupling is the Modulation Index (MI) method. The core idea of this method, similar to the first group of methods (e.g., PLI family), is based on asymmetry or a series of significant differences. When we aim to examine the existence of phase-amplitude coupling, we can assume that for a specific series, such coupling exists if an asymmetry is present.

Specific amplitudes increase or decrease depending on the phase. For example, if the phase is π, the amplitude always reaches its maximum value, and if the phase is $-\pi$, it reaches its minimum. This means that the amplitude is distributed based on its phase component. If pairs are defined as follows:

$$[A_a(t), \phi_p(t)]$$

where the amplitude and phase are aligned at each time sample, the expectation is that the points on the X-Y plane will form a distribution with a non-zero coupling coefficient (clearly asymmetric).

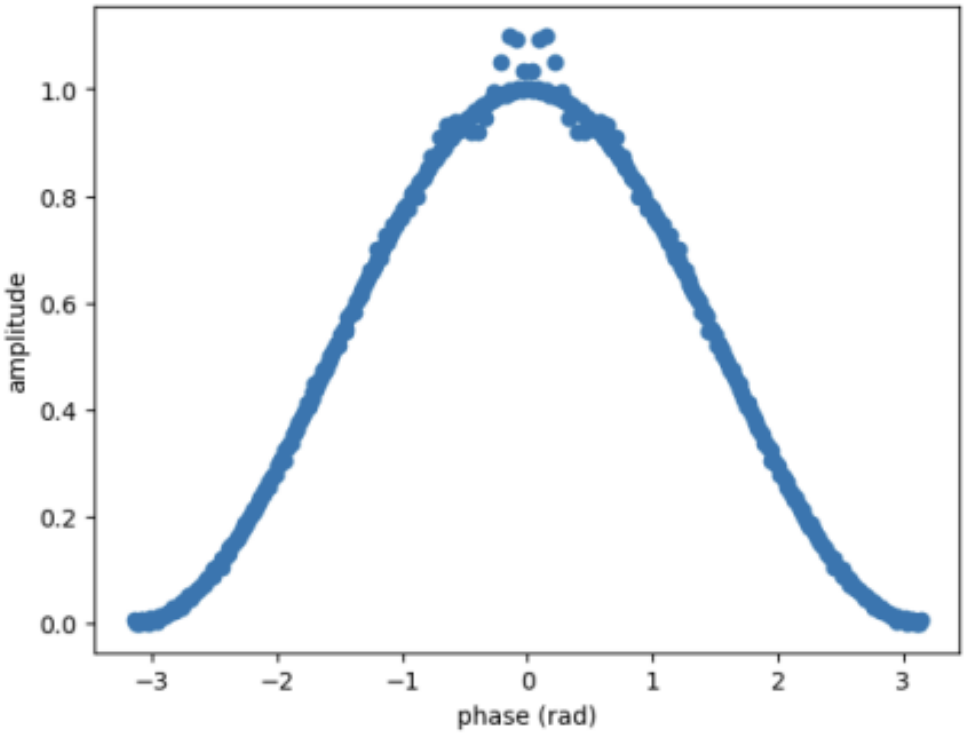
The general idea of this method is based on this asymmetry in distribution. Large amplitudes tend to be biased towards certain phases, while smaller amplitudes tend towards others. A simple example of such behavior can be shown in the following graph, where each time sample demonstrates the relationship between the amplitude and the phase.

More specifically, in this method, phase-amplitude pairs are formed at each time sample, and phases are grouped into bins. Assume we have 10 time points and want to perform this process over the range $[-\pi, \pi]$. For simplicity, divide the range into three bins: $[-\pi, -\pi/3]$, $[-\pi/3, \pi/3]$, and $[\pi/3, \pi]$.

Points include the following:

Phase	$\pi/9$	$4\pi/7$	π	$-\pi/8$	$7\pi/11$	0	$-2\pi/5$	$-3\pi/4$	$-5\pi/16$	$\pi/12$
Amplitude	0.8	0.25	0.1	0.7	0.6	0.95	0.4	0.14	0.65	0.9

Now we need to check which bin each phase point falls into. The bins are numbered in ascending order. Using this, the table can be updated with bin numbers:



Phase	$\pi/9$	$4\pi/7$	π	$-\pi/8$	$7\pi/11$	0	$-2\pi/5$	$-3\pi/4$	$-5\pi/16$	$\pi/12$
Amplitude	0.8	0.25	0.1	0.7	0.6	0.95	0.4	0.14	0.65	0.9
Bin Number	2	3	3	2	3	2	1	1	2	2

In this case, the amplitude spectrum is represented based on its corresponding bins. This means that each bin's total amplitude sums can be calculated.

Bin Number	Sum of Amplitudes	Normalized Amplitude
1	0.89	0.17
2	3.35	0.65
3	0.95	0.18

The second column indicates the total amplitude in each bin, and the last column shows the percentage of the total amplitude allocated to each phase bin.
Assume that a function, which takes a set containing an arbitrary number of points (like in the example) and maps it to the last column of the table above, is called the `ProxyHistogram` function. Essentially, this function calculates the histogram of one component of a point (e.g., the phase component of points [phase, amplitude]).
For each bin in the histogram, instead of counting the number of points that fall within that bin, the function sums the second component (in this case, the amplitude) of the points whose first component (in this case, the phase) falls within the bin. Finally, it normalizes these values by dividing by the total sum so that the resulting values add up to 1, allowing for a probabilistic interpretation.

Implement the `ProxyHistogram` function, which takes as input a set of N points in two dimensions, along with the number of bins. It outputs the values for each bin (similar to the third column in the table above).

```
    def ProxyHistogram(phi, A, num_bins):
    """
    Create a proxy histogram where the bins are defined based on `phi`,
    and the values in each bin are summed amplitudes `A`.

    Parameters:
    ----------
    phi : 1D numpy.ndarray
        Array of phase values (in radians). Shape: (time_points,)
    A : 1D numpy.ndarray
        Array of amplitude values. Shape: (time_points,)
    num_bins : int
        Number of bins to divide the phase range [-pi, pi] into.

    Returns:
    -------
    histogram : 1D numpy.ndarray
        Normalized histogram where each bin value is the summed amplitude
        of phases falling into that bin, divided by the total amplitude sum.
    bin_edges : 1D numpy.ndarray
        Edges of the bins used for the histogram.
    """
    bin_edges = np.linspace(-np.pi, np.pi, num_bins + 1)

    histogram = np.zeros(num_bins)

    for i in range(len(phi)):
        bin_index = np.digitize(phi[i], bin_edges) - 1
        if 0 <= bin_index < num_bins:
            histogram[bin_index] += A[i]

    total_sum = np.sum(histogram)
    if total_sum > 0:
        histogram = histogram / total_sum

    return histogram, bin_edges
```

- You can implement the function specifically for the phase and amplitude. In this case, the output bins always consider the range $[-\pi, \pi]$ as the phase domain.

Consider two Gaussian white noise signals. Plot the output of this function for the amplitude and phase frequency domains. Can this method reveal any connectivity between the signals?

```
    xa = np.random.normal(0, 1, len(t))
    xp = np.random.normal(0, 1, len(t))

    phi_flatten, A_flatten = calculate_phase_amp_1D(xa, xp, fs)

    phi_flatten = phi_flatten.flatten()
    A_flatten = A_flatten.flatten()

    num_bins = 3
    histogram, bin_edges = ProxyHistogram(phi_flatten, A_flatten, num_bins)

    bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2

    plt.figure(figsize=(12, 10))
```

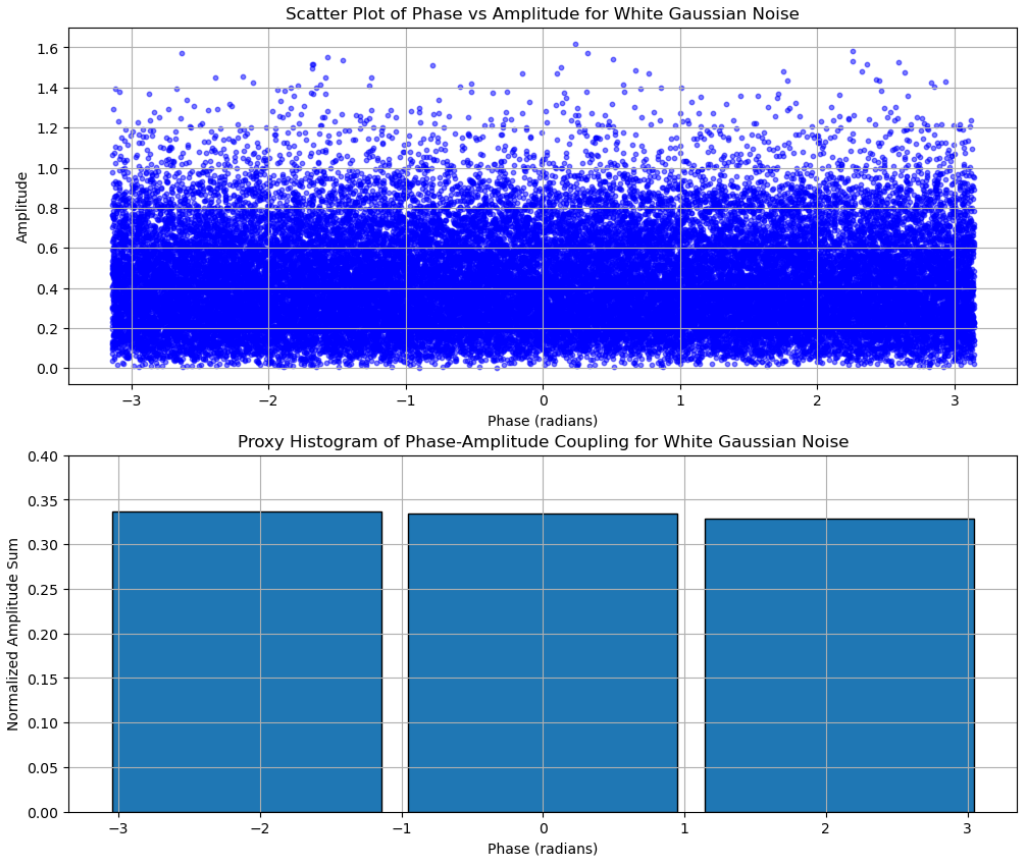


Figure 9: Output of ProxyHistogram for 2 noise

```
15
16      plt.subplot(2, 1, 1)
17      plt.scatter(phi_flatten, A_flatten, s=10, alpha=0.5, color='blue')
18      plt.title('Scatter Plot of Phase vs Amplitude for White Gaussian Noise')
19      plt.xlabel('Phase (radians)')
20      plt.ylabel('Amplitude')
21      plt.grid(True)
22
23      plt.subplot(2, 1, 2)
24      plt.bar(bin_centers, histogram, width=np.diff(bin_edges)/1.1, edgecolor='
        black', align='center')
25      plt.ylim([0,0.4])
26      plt.title('Proxy Histogram of Phase-Amplitude Coupling for White Gaussian
        Noise')
27      plt.xlabel('Phase (radians)')
28      plt.ylabel('Normalized Amplitude Sum')
29      plt.grid(True)
30      plt.show()
```

Observations

- The scatter plot in Figure 9 (top) shows the phase versus amplitude distribution for white Gaussian noise. The points are randomly scattered, indicating no specific relationship between phase and amplitude.

- The Proxy Histogram in Figure 9 (bottom) demonstrates a uniform distribution of normalized amplitude sums across all phase bins.
- The uniformity in the histogram indicates that there is no phase-amplitude coupling in the case of white Gaussian noise.
- Therefore, no significant relationship or dependency can be inferred between the phase and amplitude for these signals.

As mentioned earlier, in the absence of connectivity, the expected distribution of the amplitude sums over the frequency bins should have a uniform distribution. This means that around a frequency range, the bin sizes should not vary significantly. This allows us to assume that in the absence of connectivity, the distribution forms a uniform pattern, and deviations from uniformity indicate differences between the bins.
Therefore, comparing the phase-amplitude distribution (output of the `ProxyHistogram` function) against a uniform distribution is a suitable measure of non-uniformity. One way to compare distributions is using a metric that resembles a distance. The Kullback-Leibler Divergence metric (relation below) measures the difference between two distributions:

$$D_{KL}(P \parallel Q) = \sum_{x} P(x) \log \frac{P(x)}{Q(x)}$$

Implement this function.

- The function should take two vectors as input, representing the observed and reference distributions, and compute their divergence.

```
def ProxyHistogram(phi, A, num_bins):
    """
    Create a proxy histogram where the bins are defined based on `phi`,
    and the values in each bin are summed amplitudes `A`.

    Parameters:
    ----------
    phi : 1D numpy.ndarray
        Array of phase values (in radians). Shape: (time_points,)
    A : 1D numpy.ndarray
        Array of amplitude values. Shape: (time_points,)
    num_bins : int
        Number of bins to divide the phase range [-pi, pi] into.

    Returns:
    -------
    histogram : 1D numpy.ndarray
        Normalized histogram where each bin value is the summed amplitude
        of phases falling into that bin, divided by the total amplitude sum.
    bin_edges : 1D numpy.ndarray
        Edges of the bins used for the histogram.
    """
    bin_edges = np.linspace(-np.pi, np.pi, num_bins + 1)

    histogram = np.zeros(num_bins)

```

```
    for i in range(len(phi)):
        bin_index = np.digitize(phi[i], bin_edges) - 1
        if 0 <= bin_index < num_bins:
            histogram[bin_index] += A[i]

    total_sum = np.sum(histogram)
    if total_sum > 0:
        histogram = histogram / total_sum

    return histogram, bin_edges
```

Generate two Gaussian white noise distributions, compute their outputs using the `ProxyHistogram` function, and compare the distributions (observed and uniform) using this function. Report your findings.

```
np.random.seed(42)
gaussian_1 = np.random.normal(0, 1, 1000)  # Gaussian distribution N(0, 1)
gaussian_2 = np.random.normal(1, 1, 1000)  # Gaussian distribution N(1, 1)

uniform_1 = np.random.uniform(-1, 1, 1000)  # Uniform distribution U(-1, 1)
uniform_2 = np.random.uniform(0, 2, 1000)   # Uniform distribution U(0, 2)

kl_gaussian_same = kl_divergence(gaussian_1, gaussian_1)
kl_uniform_same = kl_divergence(uniform_1, uniform_1)

kl_gaussian_diff = kl_divergence(gaussian_1, gaussian_2)
kl_uniform_diff = kl_divergence(uniform_1, uniform_2)

# Display results
print("KL Divergence Between Homogeneous Distributions:")
print(f"Gaussian (N(0,1) vs N(0,1)): {kl_gaussian_same:.4f}")
print(f"Uniform (U(-1,1) vs U(-1,1)): {kl_uniform_same:.4f}")

print("\nKL Divergence Between Heterogeneous Distributions:")
print(f"Gaussian (N(0,1) vs N(1,1)): {kl_gaussian_diff:.4f}")
print(f"Uniform (U(-1,1) vs U(0,2)): {kl_uniform_diff:.4f}")
```

Soloution

KL Divergence Between Homogeneous Distributions:

- Gaussian (N(0,1) vs N(0,1)): 0.0000
- Uniform (U(-1,1) vs U(-1,1)): 0.0000

KL Divergence Between Heterogeneous Distributions:

- Gaussian (N(0,1) vs N(1,1)): 0.8806
- Uniform (U(-1,1) vs U(0,2)): 9.1953

Up to this point, by having appropriate time samples of phase and amplitude from the signals, we can define a numerical metric for phase-amplitude coupling.
Implement the final function for `Modulation Index (MI)` as described below:

- The input to your function is the phase matrix $\phi(t, f_p)$ and the amplitude matrix $A(t, f_a)$

from the previous step.

- Using the `ProxyHistogram` function, compute the distribution of amplitudes across phase bins for each frequency range.
- In the final step, compare the resulting distribution from the previous step to a uniform distribution using the Kullback-Leibler divergence, then divide the result by $\log K$, where K is the number of phase bins. The result is the phase-amplitude coupling value.

```
def kl_divergence(P_samples, Q_samples, num_bins=50):
    """
    Compute the Kullback-Leibler Divergence D_KL(P || Q) between two
    distributions.

    Parameters:
    ----------
    P_samples : array-like
        Samples from distribution P.
    Q_samples : array-like
        Samples from distribution Q.
    num_bins : int, optional
        Number of bins to use for the histograms (default is 50).

    Returns:
    -------
    float
        The KL divergence D_KL(P || Q).
    """
    # Compute histograms for P and Q
    hist_P, bin_edges = np.histogram(P_samples, bins=num_bins, density=True)
    hist_Q, _ = np.histogram(Q_samples, bins=bin_edges, density=True)

    # Normalize histograms to get probabilities
    P = hist_P / np.sum(hist_P)
    Q = hist_Q / np.sum(hist_Q)

    # Ensure no zeros in Q for division (replace with small epsilon)
    Q = np.where(Q == 0, 1e-10, Q)
    P = np.where(P == 0, 1e-10, P)  # Handle P to avoid log(0)

    # Compute KL divergence
    kl_div = np.sum(P * np.log(P / Q))

    return kl_div

def calculate_pac_MI(phi, A, num_bins):
    """
    Calculate the PAC (Modulation Index) matrix for given phi and A arrays.

    Parameters:
    ----------
    phi : 2D numpy.ndarray
        Phase array of shape (time_points, phase_freqs)
    A : 2D numpy.ndarray
        Amplitude array of shape (time_points, amp_freqs)
    num_bins : int
        Number of bins to use in the ProxyHistogram.
```

```
    Returns:
    -------
    PAC : 2D numpy.ndarray
        PAC matrix of shape (amp_freqs, phase_freqs), representing the MI for each combination.
    """
    _, phase_freqs = phi.shape
    _, amp_freqs = A.shape

    PAC = np.zeros((amp_freqs, phase_freqs))

    U = np.ones(num_bins) / num_bins

    for f_a in range(amp_freqs):
        for f_p in range(phase_freqs):
            phi_1d = phi[:, f_p]
            A_1d = A[:, f_a]

            P, _ = ProxyHistogram(phi_1d, A_1d, num_bins)

            eps = 1e-10
            P = np.where(P <= 0, eps, P)

            D_KL_UP = np.sum(U * np.log(U / P))

            PAC[f_a, f_p] = D_KL_UP / np.log(num_bins)

    return PAC
```

The main formula for calculating PAC using MI is given as:

$$PAC = \frac{D_{KL}(U \parallel P)}{\log K}$$

Here, K is the number of bins, U is the uniform distribution, and P is the normalized distribution of amplitudes across phase bins.

Now, using the functions implemented earlier, generate the signals $x_a(t)$ and $x_p(t)$ with the following parameters:

$$k_p = 1,\ k_a = 1,\ f_{\text{phase}} = 5\,\text{Hz},\ f_{\text{amp}} = 60\,\text{Hz},\ \chi = 0,\ \sigma_n = 0.5$$

➖ *Different Values of* χ

For χ values of $\{0, 0.5, 1\}$, calculate the PAC values using both the `MI` and `MVL` methods and plot the heatmap for the PAC values over phase and amplitude frequencies. This is referred to as the comodulogram.

```
kp = 1
ka = 1
fp = 5
f_amp = 60
chi_values = [0, 0.5, 1]
sigma = 0.5

PAC_results = {}

```

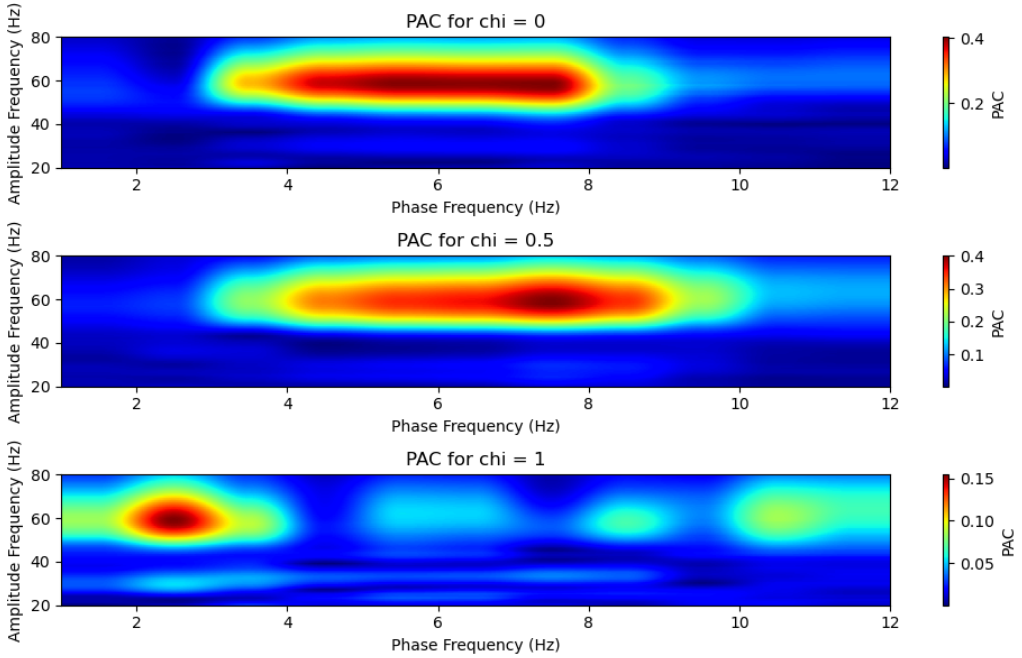


Figure 10: PAC with MLV method for different value of χ

```
    for chi in chi_values:
        t, xp, xa = generate_signals(kp, ka, fp, f_amp, chi, sigma)

        phi, A = calculate_phase_amp(xa, xp, fs)

        PAC = calculate_pac_MLV(phi.T, A.T)
        PAC = calculate_pac_MI(phi.T, A.T, num_bins)

        PAC_results[chi] = PAC

    plt.figure(figsize=(10, 6))

    for idx, chi in enumerate(chi_values):
        plt.subplot(3, 1, idx + 1)
        plt.imshow(PAC_results[chi], aspect='auto', origin='lower', cmap='jet'
, extent=[1, 12, 20, 80])
        plt.colorbar(label='PAC')
        plt.title(f'PAC for chi = {chi}')
        plt.xlabel('Phase Frequency (Hz)')
        plt.ylabel('Amplitude Frequency (Hz)')

    plt.tight_layout()
    plt.show()
```

- Based on the results, what does the parameter χ indicate?

Observations

The parameter χ determines the strength of phase-amplitude coupling (PAC) between the phase and amplitude components of the signal.

- In the ideal case where $\chi = 1$ and $\chi = 0$, what do you expect the comodulograms to look like?

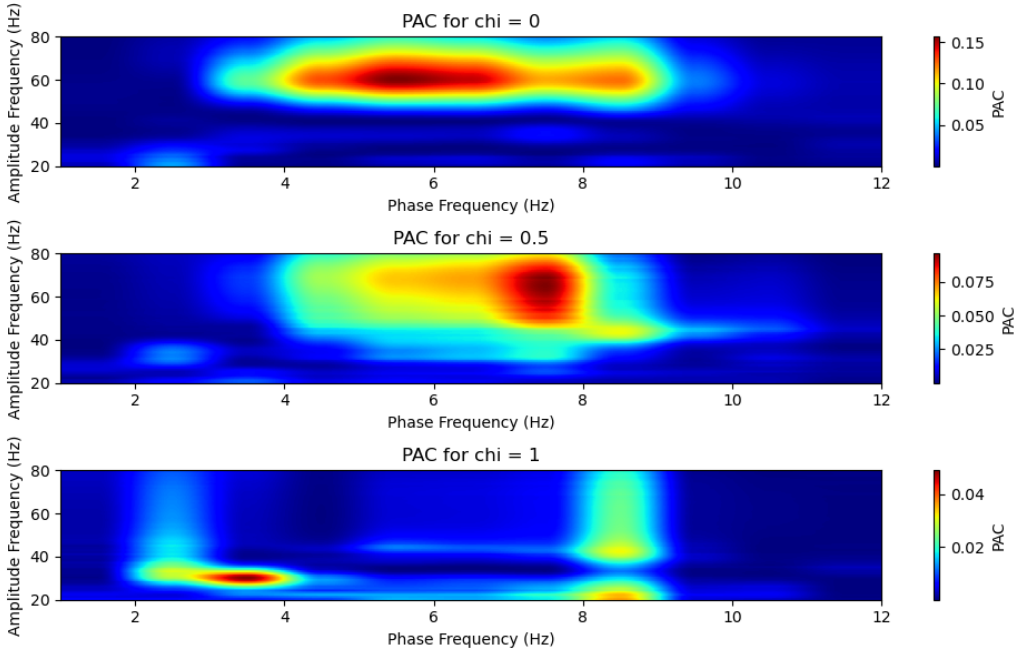


Figure 11: PAC with MI method for different value of χ

Observations

- **For $\chi = 0$:**
  - No phase-amplitude coupling (PAC) is expected.
  - The comodulogram should appear uniform with no specific patterns or peaks, as the phase and amplitude frequencies are independent.
  - PAC values should be close to zero across all regions of the plot.
- **For $\chi = 1$:**
  - Strong phase-amplitude coupling is expected, but dominated entirely by one frequency component.
  - The comodulogram should show a single, localized peak where the phase and amplitude frequencies align, indicating a deterministic relationship.
  - Outside of this localized region, PAC values should be near zero.

- Which PAC calculation method works better?

Observations

The MI method performs better as it provides more detailed and interpretable results, particularly in identifying localized phase-amplitude coupling regions.

MI (Modulation Index):

- Provides a more nuanced and detailed representation of PAC.

- Shows clearer differences in PAC values for varying χ, making it easier to interpret the relationship between phase and amplitude.
- Highlights the localized coupling regions effectively, particularly for $\chi = 0.5$ and $\chi = 1$.

MLV (Mean Vector Length):

- Captures the overall coupling strength but lacks finer details.
- For $\chi = 0$, $\chi = 0.5$, and $\chi = 1$, it provides less separation between PAC levels, making the interpretation less precise.
- The results appear more uniform, which might obscure specific coupling patterns.

Different Values of σ_n

Now, keeping $\chi = 1$ fixed, compare the comodulograms for both methods under increasing noise levels $\sigma_n = \{0.5, 1, 2, 3\}$.

```
kp = 1
ka = 1
fp = 5
f_amp = 60
chi = 0
sigma_values = [0.5, 1, 2, 3]

PAC_results_sigma = {}

for sigma in sigma_values:
    t, xp, xa = generate_signals(kp, ka, fp, f_amp, chi, sigma)

    phi, A = calculate_phase_amp(xp, xa,fs)

    PAC = calculate_pac_MLV(phi.T, A.T)
    PAC = calculate_pac_MI(phi.T, A.T, num_bins)

    PAC_results_sigma[sigma] = PAC

plt.figure(figsize=(10, 8))

for idx, sigma in enumerate(sigma_values):
    plt.subplot(4, 1, idx + 1)
    plt.imshow(PAC_results_sigma[sigma], aspect='auto', origin='lower', cmap='jet', extent=[1, 12, 20, 80])
    plt.colorbar(label='PAC')
    plt.title(f'PAC for sigma_n = {sigma}')
    plt.xlabel('Phase Frequency (Hz)')
    plt.ylabel('Amplitude Frequency (Hz)')

plt.tight_layout()
plt.show()
```

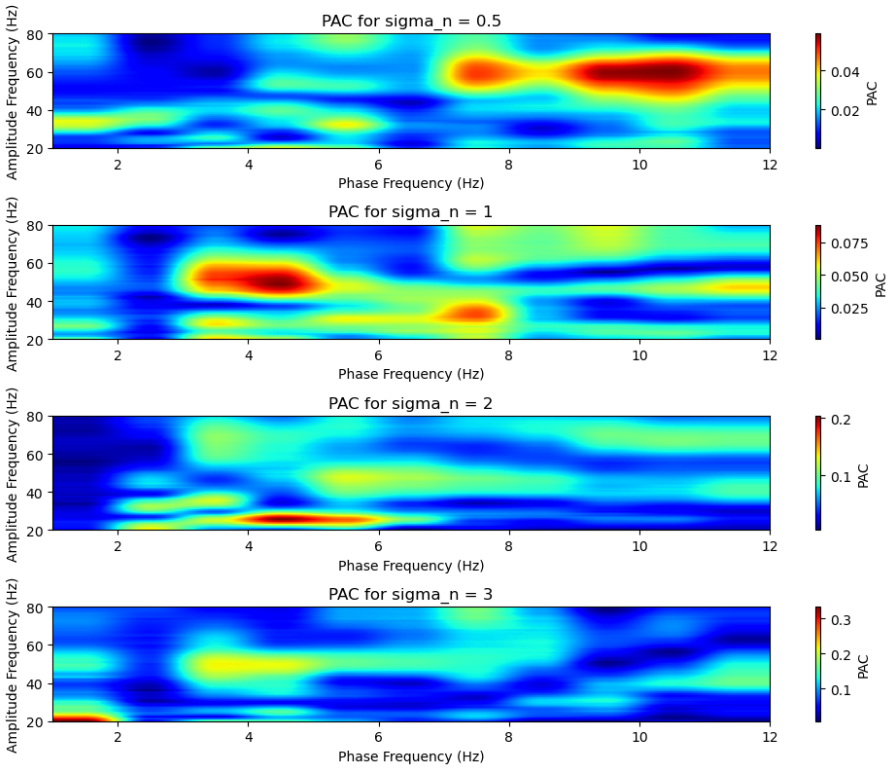


Figure 12: PAC with MLV method for different value of σ_n

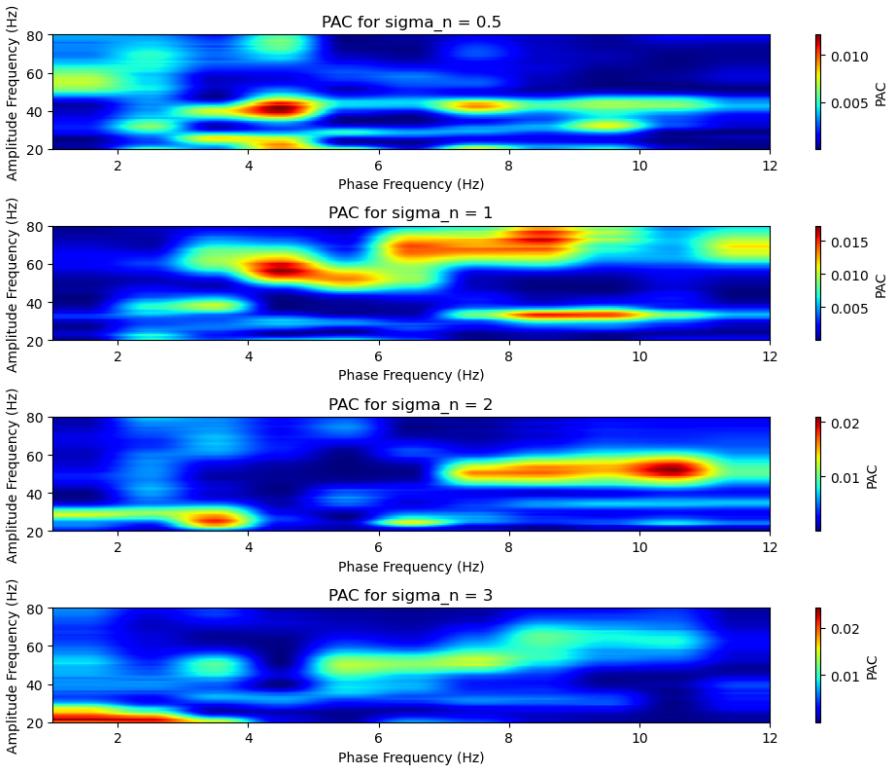


Figure 13: PAC with MI method for different value of σ_n

Observations

MLV Method:

- At $\sigma_n = 0.5$, clear PAC patterns with strong localized peaks.
- As noise increases, PAC values become less distinct, and at $\sigma_n = 3$, patterns are highly distorted.
- MLV excels in low noise but is highly sensitive to noise.

MI Method:

- At $\sigma_n = 0.5$, PAC patterns are visible but less pronounced compared to MLV.
- More robust to noise, retaining structured PAC patterns even at $\sigma_n = 2$ and $\sigma_n = 3$.
- MI is better at handling noisy environments, providing consistent results at higher σ_n.

- Which PAC calculation method works better under increasing noise levels?

Observations

The MI method performs better against increased noise power. It maintains structured PAC patterns even at high noise levels ($\sigma_n = 2$ and $\sigma_n = 3$). In contrast, the MLV method is highly sensitive to noise, with PAC patterns degrading significantly as noise increases and becoming indistinct at $\sigma_n = 3$. Therefore, the MI method is more robust and reliable in noisy environments.

▬ *Circular Shift*

Finally, apply a circular shift (e.g., 100 ms) to the x_p signal and recompute the comodulograms for the x_a and x_p signals ($\chi = 0$, $\sigma_n = 0.5$).

```
kp = 1
ka = 1
fp = 5
f_amp = 60
chi = 0
sigma = 0.5

t, xp, xa = generate_signals(kp, ka, fp, f_amp, chi, sigma)

phi, A = calculate_phase_amp(xp, xa, fs)
PAC_original = calculate_pac_MLV(phi.T, A.T)
PAC_original = calculate_pac_MI(phi.T, A.T, num_bins)

shift_samples = int(0.1 * 500)  # fs = 500 Hz, so shift by 100 ms

xp_shifted = circular_shift(xp, shift_samples)
xa_shifted = circular_shift(xa, shift_samples)

phi_shifted, A_shifted = calculate_phase_amp(xp_shifted, xa_shifted, fs)
PAC_shifted = calculate_pac_MLV(phi_shifted.T, A_shifted.T)
PAC_shifted = calculate_pac_MI(phi_shifted.T, A_shifted.T, num_bins)
```

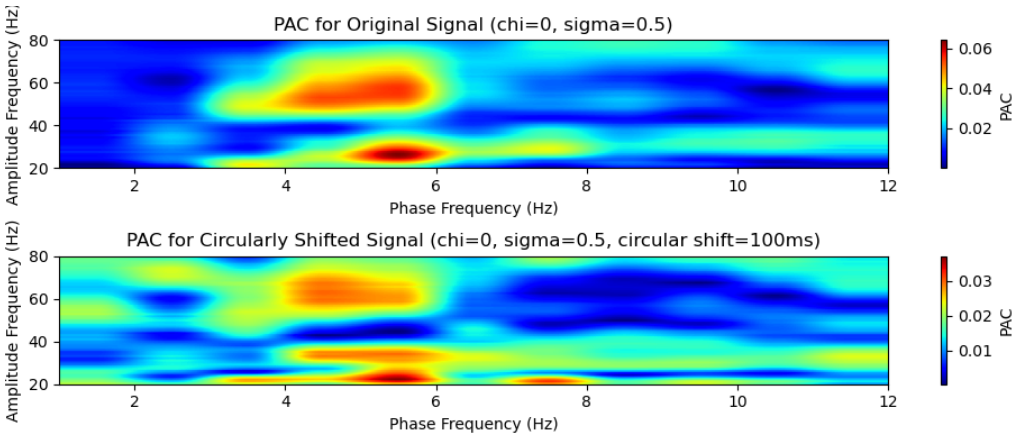


Figure 14: PAC with MLV method for circular shift

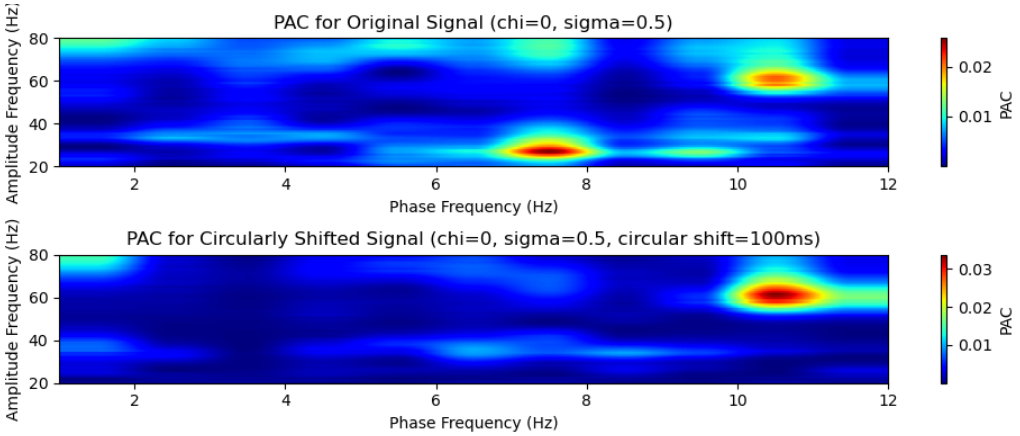


Figure 15: PAC with MI method for circular shift

```
22
23      plt.figure(figsize=(10, 4))
24
25      plt.subplot(2, 1, 1)
26      plt.imshow(PAC_original, aspect='auto', origin='lower', cmap='jet', extent
        =[1, 12, 20, 80])
27      plt.colorbar(label='PAC')
28      plt.title('PAC for Original Signal (chi=0, sigma=0.5)')
29      plt.xlabel('Phase Frequency (Hz)')
30      plt.ylabel('Amplitude Frequency (Hz)')
31
32      plt.subplot(2, 1, 2)
33      plt.imshow(PAC_shifted, aspect='auto', origin='lower', cmap='jet', extent
        =[1, 12, 20, 80])
34      plt.colorbar(label='PAC')
35      plt.title('PAC for Circularly Shifted Signal (chi=0, sigma=0.5, circular
        shift=100ms)')
36      plt.xlabel('Phase Frequency (Hz)')
37      plt.ylabel('Amplitude Frequency (Hz)')
38
39      plt.tight_layout()
40      plt.show()
```

- Based on the results, can we consider PAC a reliable metric for delay invariance?

Observations

For both methods (MLV and MI), PAC patterns change after applying a circular shift of 100 ms. The localized peaks shift or reduce, indicating that PAC is sensitive to phase shifts or delays. Therefore, PAC cannot be considered fully resistant to delay effects, as the patterns depend on the alignment of phase and amplitude components. However, the MI method shows slightly more stability compared to the MLV method.

▬ *Common Source Effect*

In the following section, similar to previous sections, we evaluate the effect of a common source and assess the performance of each PAC calculation method when dealing with the impact of a common source.

First, generate the signals x_p and x_a with the following parameters:

$$k_p = 1,\ k_a = 1,\ f_{\text{phase}} = 5\,\text{Hz},\ f_{\text{amp}} = 60\,\text{Hz},\ \chi = 0,\ \sigma_n = 0.5$$

Then, add a sinusoidal term to each of the two signals as follows:

$$\hat{x}_p = x_p + \alpha\cos(2\pi f_{\text{cs}} t)$$

$$\hat{x}_a = x_a + \beta\cos(2\pi f_{\text{cs}} t)$$

where $f_{\text{cs}} = 40\,\text{Hz}$.

For various values of $\alpha = \beta = \{0, 5, 10, 50\}$, calculate PAC values for the signals $\hat{x}_p$ and $\hat{x}_a$ using both the `MI` and `MVL` methods. Plot the resulting comodulograms and compare the results.

```
kp = 1
ka = 1
fp = 5
f_amp = 60
chi = 0
sigma = 0.5
f_cs = 40

alpha_values = [0, 5, 10, 50]
beta_values = [0, 5, 10, 50]

results_PAC_MI = {}
results_PAC_MLV = {}

t, xp, xa = generate_signals(kp, ka, fp, f_amp, chi, sigma)

for alpha in alpha_values:
    for beta in beta_values:

        xp += alpha * np.cos(2 * np.pi * f_cs * t)
        xa += beta * np.cos(2 * np.pi * f_cs * t)

        phi, A = calculate_phase_amp(xp, xa, fs)

        PAC_MLV = calculate_pac_MLV(phi.T, A.T)
        PAC_MI = calculate_pac_MI(phi.T, A.T, num_bins)

        results_PAC_MLV[(alpha, beta)] = PAC_MLV
```

```
            results_PAC_MI[(alpha, beta)] = PAC_MI


    for alpha in alpha_values:
        for beta in beta_values:
            plt.figure(figsize=(10, 4))
            plt.subplot(2, 1, 1)
            plt.imshow(results_PAC_MI[(alpha, beta)], aspect='auto', origin='
    lower', cmap='jet')
            plt.title(f'MI PAC (alpha={alpha}, beta={beta})')
            plt.colorbar(label='PAC')
            plt.xlabel('Phase Frequency (Hz)')
            plt.ylabel('Amplitude Frequency (Hz)')

            plt.subplot(2, 1, 2)
            plt.imshow(results_PAC_MLV[(alpha, beta)], aspect='auto', origin='
    lower', cmap='jet')
            plt.title(f'MVL PAC (alpha={alpha}, beta={beta})')
            plt.colorbar(label='PAC')
            plt.xlabel('Phase Frequency (Hz)')
            plt.ylabel('Amplitude Frequency (Hz)')

            plt.tight_layout()
            plt.show()
```

Observations

MI Method:

- PAC is localized and highlights strong coupling for higher values of α and β.
- The method clearly differentiates the impact of α and β, with PAC increasing as their values rise.
- For $\alpha = 0$ or $\beta = 0$, PAC is minimal, indicating the absence of coupling.
- The MI method is more precise and highlights the influence of α and β more effectively.

MLV Method:

- PAC values are generally higher compared to MI, but the method is less specific and more spread out.
- The PAC patterns are broader, and it becomes harder to isolate the contributions of α and β.
- For high α and β, coupling regions are more prominent but less localized than in MI.
- The MLV method produces stronger but less specific coupling patterns, making it less suitable for isolating the effects of individual parameters.

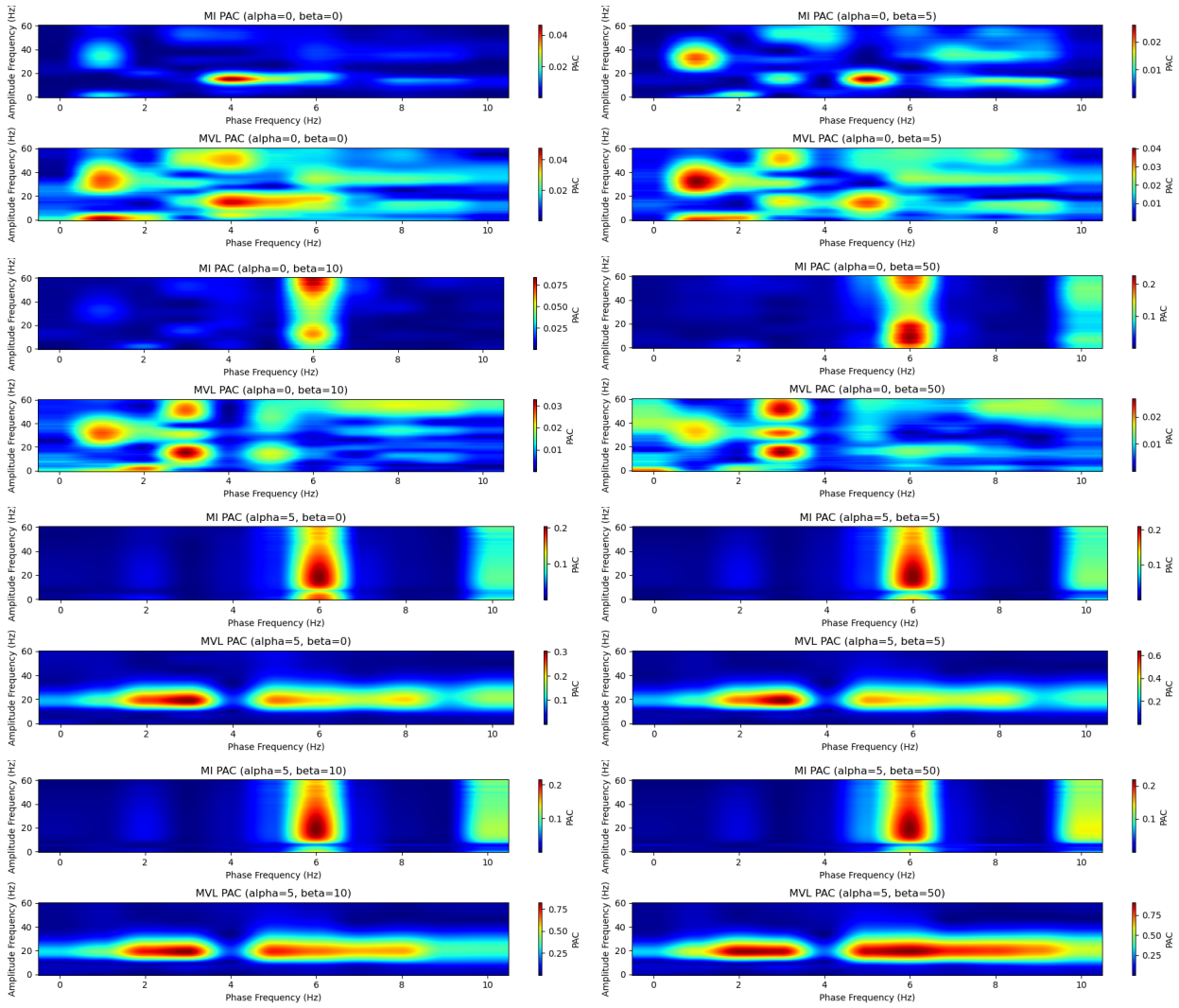


Figure 16: $\alpha = 0, 5 \quad \beta = 0, 5, 10, 50$

- What effect do increasing α and β have on the results?

Observations

Increasing the values of α and β has the following effects on the PAC results:

- **PAC Magnitude:** Both MI and MLV methods show higher PAC values as α and β increase. This indicates stronger phase-amplitude coupling due to the addition of sinusoidal components with higher weights.
- **Localization (MI Method):** For the MI method, the PAC becomes more localized in specific phase and amplitude frequency regions, reflecting a more focused coupling.
- **Spread (MLV Method):** For the MLV method, PAC patterns become broader but stronger. This shows less precise coupling but indicates an overall increase in coupling strength.
- **Interaction of α and β:** Interaction between α and β becomes more prominent at higher values, with PAC reflecting the combined effect of the two parameters.

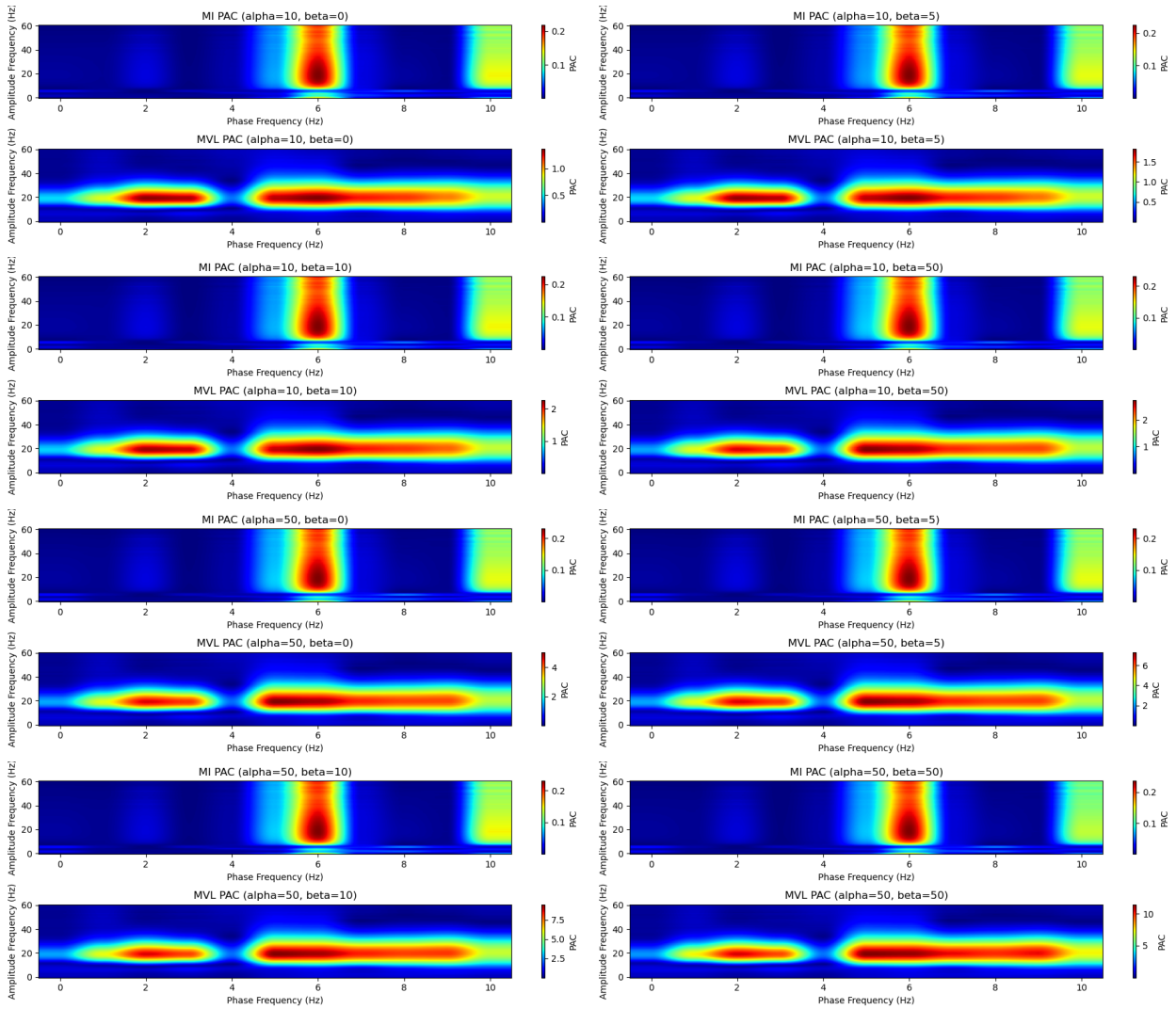


Figure 17: $\alpha = 10, 50 \quad \beta = 0, 5, 10, 50$

- Which method performs better in addressing the effect of a common source?

Observations

The MI method performs better against the effect of a common source due to its ability to localize PAC patterns and differentiate between true coupling and noise or shared source effects. MI shows more precise and structured results, even when α and β are high, indicating better resistance to the influence of a common source. The MLV method, on the other hand, produces broader and less distinct PAC patterns, making it harder to isolate the specific effects of a common source. This makes MLV less reliable in scenarios where a common source might introduce artifacts.
MI is more robust and effective in distinguishing true coupling from the effects of a common source.

Comparison of Connectivity Metrics

Considering the metrics introduced in Part One, they evaluate and measure different aspects of connectivity, but it is not possible to rank them strictly against one another. However, we can assess them under certain assumptions to determine which metrics indicate the existence of connectivity (or lack thereof) better.

On this basis, three introduced metrics should be compared under the following considerations:

- The effect of delay between signals.
- The effect of a common source on the signals.
- The effect of noise on the signals.

Perform the comparison and provide your findings.
In fact, you need to write a coherent response to the following questions:

- In the presence of delays in coupled oscillations, which method is less affected?
- Does a common source create false couplings in any method?
- Which method is more resistant to noise at low SNRs?
- What factors contribute to high-quality results in these methods?

Soloution

In the study of phase-amplitude coupling (PAC) using the MI and MLV methods, a detailed comparison reveals significant insights into their respective performance under varying conditions, including delays, common source effects, and noise. This analysis provides a comprehensive understanding of how these methods behave and their suitability for different scenarios.

1. In the presence of delays in coupled oscillations, which one is less affected?
First, in the presence of delays in coupled oscillations, the MI method demonstrates clear superiority. It maintains consistent PAC patterns even when circular shifts are introduced, showing its robustness against phase misalignment. This consistency allows researchers to trust the MI method for identifying genuine coupling relationships, even when delays occur naturally in signals. Conversely, the MLV method struggles with such delays, as the circular shift significantly impacts its PAC patterns. This sensitivity to phase misalignment makes MLV less reliable in scenarios where delay effects are unavoidable.

2. Does a common source create false couplings in any method?
Second, when a common source affects the signals, the MI method again proves to be more effective. A common source often introduces false couplings, which can be mistaken for genuine interactions. The MI method minimizes these false positives and isolates true phase-amplitude interactions, providing more accurate results. On the other hand, the MLV method is more susceptible to detecting false couplings caused by shared components. This limitation occurs because MLV does not adequately distinguish between genuine signal interactions and artifacts introduced by a common source, reducing its reliability in such cases.

3. Which method is more resistant to noise at low SNRs?
Third, noise plays a critical role in PAC analysis, especially at low SNRs. In this context, the MI method outperforms the MLV method due to its resilience against noise. Even at low SNRs, the MI method retains structured PAC patterns, allowing researchers to extract meaningful insights from noisy data. In contrast, the MLV method suffers significant degradation in PAC patterns as noise increases. Its sensitivity to noise leads to a loss of specificity, making it less effective for applications in noisy environments. This robustness of MI to noise is a key factor in its preference for real-world signal analysis.

4. What factors contribute to high-quality results in these methods?
The key factors contributing to the MI method's higher performance include its precision in detecting localized PAC patterns, its ability to minimize false positives caused by noise and shared sources, and its stability across varying conditions. The MLV method, while sensitive and effective in low-noise environments, lacks the specificity and robustness needed for more challenging scenarios. It performs well when noise and common sources are not significant issues but fails to provide reliable results under complex conditions.
Overall, the MI method is a more reliable and versatile tool for PAC analysis. Its resistance to noise, delays, and shared source effects makes it particularly well-suited for real-world signal analysis where such factors are commonly present. In contrast, the MLV method's broader PAC patterns and higher sensitivity to artifacts limit its applicability in complex scenarios. Therefore, the MI method is recommended for studies requiring high accuracy and robustness in PAC detection. Its ability to handle noise, delay, and common source challenges ensures more reliable and interpretable results, making it the preferred choice for researchers.

Explanation of the Time-Frequency Wavelet Analysis Method

In part of this exercise and the exercises to follow, we need to perform time-frequency analysis. Various methods exist for time-frequency decomposition of real signals, and each has its applications. In processing biological signals, one of the most popular methods is the wavelet transform. Like other decomposition methods, the idea of this method is based on projecting the signal onto a set of reliable bases. In wavelet-based decomposition, these bases are sinusoidal and cosine signals with specific properties.

Different frequencies are analyzed; however, a group of wavelets is used, which allows us to simultaneously adjust scale and delay, adapting to different scales and delays. These functions are called mother wavelets. By scaling and translating the mother wavelet, different signal components can be extracted. The formula is as follows:

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right)$$

The wavelet transform is defined as:

$$WT\{x\}(a,b) = \int_{\mathbb{R}} x(t)\psi_{a,b}(t)\,dt$$

Here, ψ is the mother wavelet, and the result of the wavelet transform is calculated based on the delay b and scale a. These two parameters determine frequency and time.

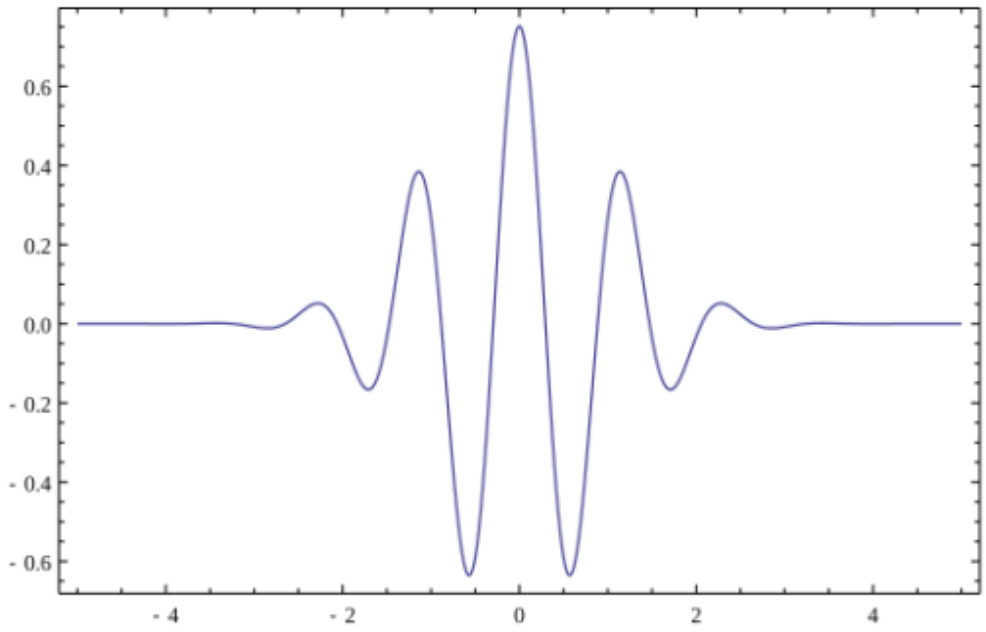


Figure 18: Morlet Wavelet

One of the commonly used wavelets is the Morlet wavelet, whose shape is shown in the image below:

The exact details of how this transform works are beyond the scope of this discussion. For now, we focus on obtaining the wavelet transform of a signal. To perform this operation in Python, you can use the `pywt.cwt` function for more detailed implementation.