Analysis of Electrophysiological Data in Neuroscience

Contents

Tutorials and scripts for analyzing electrophysiological (LFP and spike train) data.

Identifying Oscillations

This notebook will show how to quantify neural oscillations. Here, from some example data, we will calculate and plot:

1. Power spectrum
2. Windowed (classic) spectrogram
3. Wavelet based spectrogram

Frequency Domain

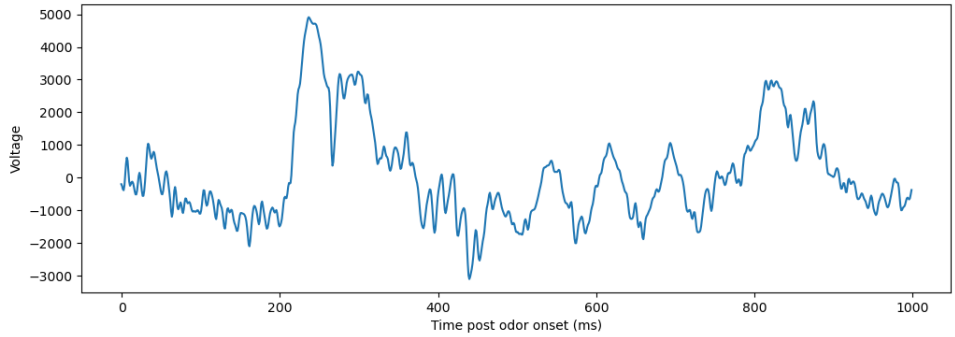
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import welch, spectrogram
```

```
#Load in the LFP, stimulation time data
fpath = (r'C:\Users\Michael\Documents\ephys_analysis_tutorials\\')
lfp_data = np.load(fpath+'example_lfp_160819_bank2.npy')
stim_times = np.load(fpath+'example_stim_times_160819_bank2.npy')
fs = 1000
```

Here, we have lfp data (already filtered + resampled to 1000Hz) from an electrode in the olfactory bulb, recording while odors are presented. The array stim_times indicates when odors are experienced (the first inhalation after each odor presentation), for 6 different odors, formatted as odors x events.

```
#Plot the lfp data in response to stimulation
evt = stim_times[0,7] #Look at the first presentation of the first odor
time_segment = [int(evt*fs), int(evt*fs+1000)] #start and end event times of interest,
converted to ms
plt.figure(figsize=(12,4))
plt.plot(np.arange(time_segment[0], time_segment[1], 1)-evt*fs,
lfp_data[time_segment[0]:time_segment[1]])
plt.xlabel('Time post odor onset (ms)')
plt.ylabel('Voltage')
```

```
Text(0, 0.5, 'Voltage')
```



We see a big deflection around 200ms, and an oscillation until a little after 600ms. Next, we'll want to quantify the presence of this oscillation. A common first step here is to plot the power spectrum of the data.

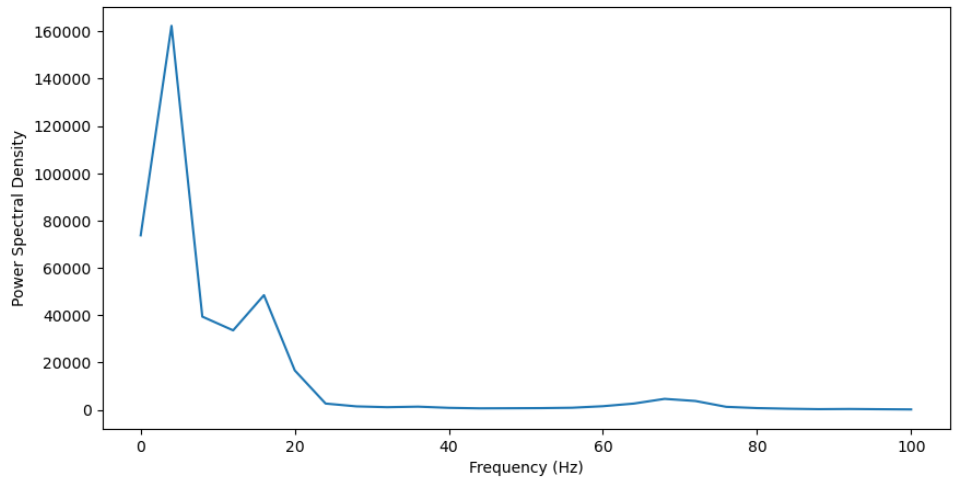
Power Spectrum

Using a Fourier transformation, any signal can be decomposed into some combination of sinusoidal oscillations. The power spectrum of a signal describes the distribution of frequency components that compose that signal. It is often used to quantify the presence of different oscillations, as a power spectrum quantifies the amplitude across frequencies.

More specifically, to compute a power spectrum, the autocorrelation of a signal is computed, and then the Fourier Transform of the result is taken. Though this can be computed once, across the entire signal, in practice this results in a noisy estimate of the power spectrum. Here, we'll use Welch's method [1] to compute power, where the signal is segmented into a series of overlapping windows, and the power spectrum of each segment are averaged. The window size and overlap are determined by the *nperseg* and *noverlap* parameters in scipy's welch function. Feel free to mess around with these to get an idea of the tradeoffs between large and small windows, and the degree of overlap.

```
f, Pxx = welch(lfp_data[time_segment[0]:time_segment[1]], fs=fs, nperseg=250,
noverlap=100)
f = f[np.where(f<=100)] #select only frequencies below 100Hz
Pxx = Pxx[np.where(f<=100)]
plt.figure(figsize=(10,5))
plt.plot(f, Pxx)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power Spectral Density')
```

```
Text(0, 0.5, 'Power Spectral Density')
```



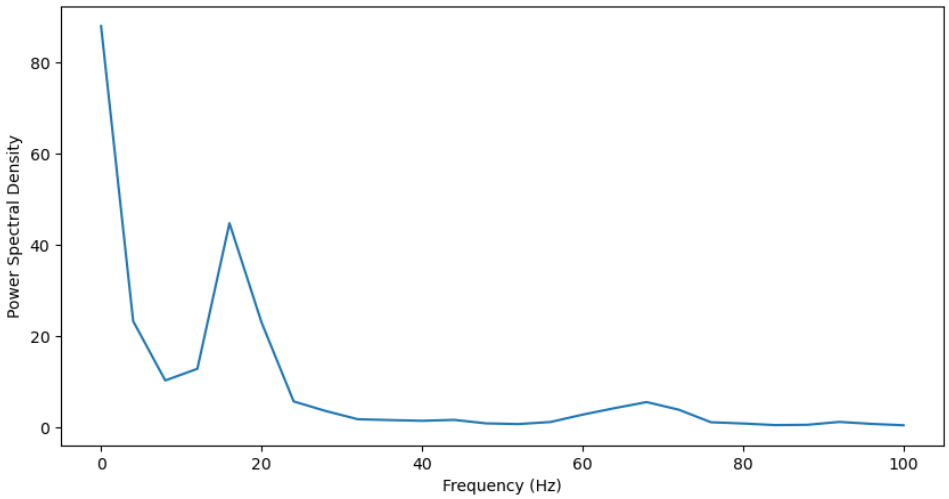
The power spectrum is much higher at lower frequencies (<20Hz). This is characteristic of neural LFP data, referred to as 1/f or pink noise, where power systematically decreases with frequency. Still, we can see three peaks in power that go beyond this trend: one around 5Hz, one just below 20Hz, and another small peak around 70Hz. To better visualize this, we should remove the 1/f noise in the data. Though an

excellent tool has been developed for this purpose, we can also remove this noise by normalizing to a baseline period, before odor presentation, where this noise will still be present. This will also help us identify which oscillations are specifically induced by the odor.

To do so, we'll take the power spectrum for 1 second of data before the odor was presented, and divide the odor-evoked psd by this baseline psd. **Note**: Make sure your parameters match the ones you used to compute the first power spectrum, or else the frequency resolution won't match and you won't be able to normalize.

```
Pxx_evoked = Pxx
time_segment = [int(evt*fs)-1000, int(evt*fs)] #start and end event times of interest,
converted to ms
f, Pxx_baseline = welch(lfp_data[time_segment[0]:time_segment[1]], fs=fs, nperseg=250,
noverlap=100)
f = f[np.where(f<=100)] #select only frequencies below 100Hz
Pxx_baseline = Pxx_baseline[np.where(f<=100)]
Pxx_normalized = Pxx_evoked / Pxx_baseline
plt.figure(figsize=(10,5))
plt.plot(f, Pxx_normalized)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power Spectral Density')
```

```
Text(0, 0.5, 'Power Spectral Density')
```



That helps a little bit: we now clearly see the peak in beta (~20Hz). There's also a big increase in power at the lowest frequencies.

A power spectra is a useful first step for examining the oscillatory content of a time-series. However, we are still missing some information. For example, here, the gamma (~70Hz) oscillation is pretty muted. One reason for this is the gamma oscillation is brief, and here we're looking at power over the entire second, making it hard to distinguish between a brief oscillation from a weak one. To better analyze this, let's look at how power changes over time.

Time Frequency Domains

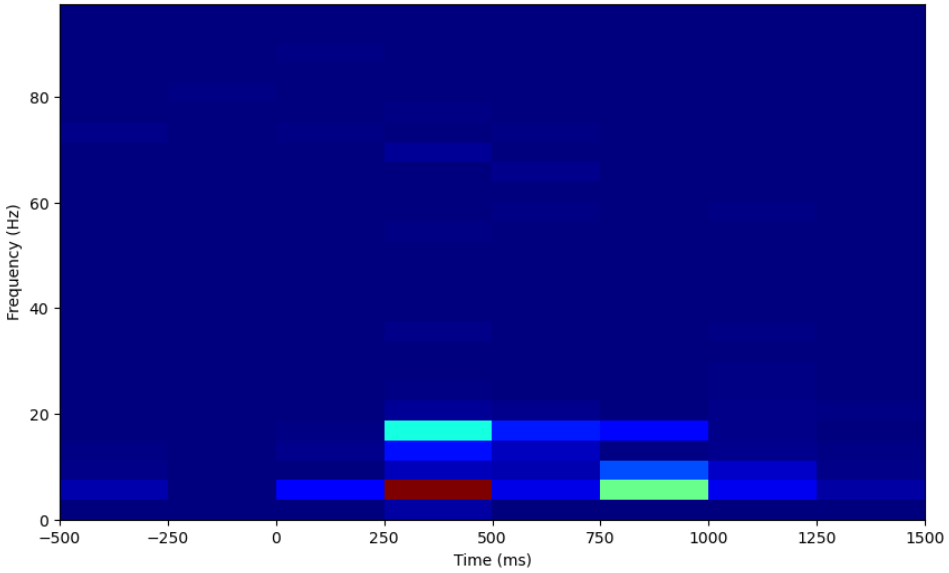
Time frequency analyses, as the name indicates, introduce a time dimension to our power analyses. This allows us to analyze when exactly oscillations are present. Now, our outputs are matrices called spectrograms, plotted as heatmaps, of power over both frequency and time.

Window analysis

A simple and common time frequency analysis is to compute a spectrogram as a series of power spectra. Here, the data is broken up into a set of overlapping windows, and the power spectrum is computed on each of these windows. We can use scipy's spectrogram function here.

```
relative_times = [-500,1500]
time_segment = [int(evt*fs)+relative_times[0], int(evt*fs+relative_times[1])] #start and
end event times of interest, converted to ms
f, t, Sxx = spectrogram(lfp_data[time_segment[0]:time_segment[1]], fs)
f = f[np.where(f<=100)] #select only frequencies below 100Hz
Sxx = Sxx[np.where(f<=100)[0],:] #select only frequencies below 100Hz
plt.figure(figsize=(10,6))
plt.imshow(Sxx, extent=[relative_times[0], relative_times[-1], f[0], f[-1]], aspect =
'auto', origin = 'lower', cmap='jet')
plt.xlabel('Time (ms)')
plt.ylabel('Frequency (Hz)')
```

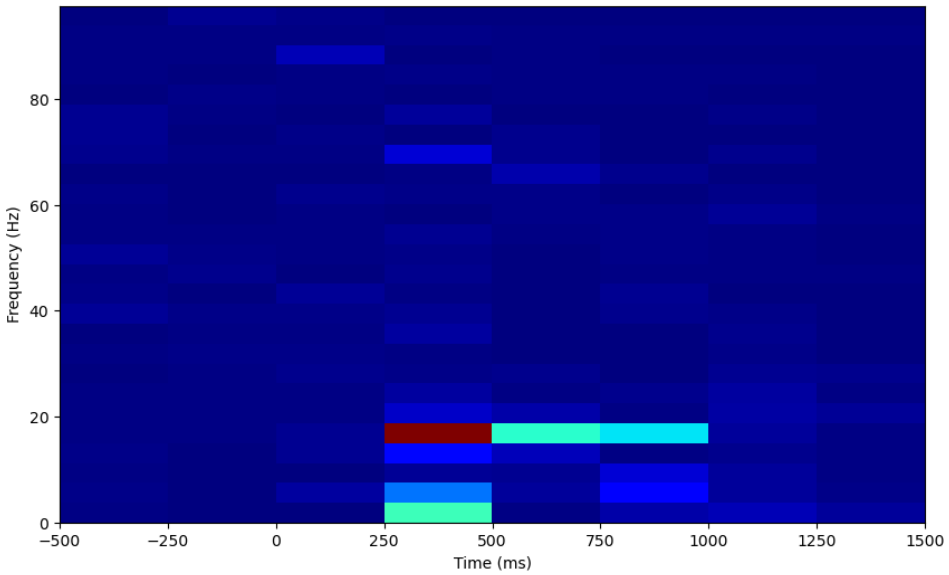
```
Text(0, 0.5, 'Frequency (Hz)')
```



Let's normalize this too. Now that we have data on how our power changes over time, we get multiple samples of power data for each frequency, and we can z-score the evoked power to this distribution at baseline.

```
Sxx_evoked = Sxx
time_segment = [int(evt*fs)-2000, int(evt*fs)] #start and end event times of interest,
converted to ms
f, t, Sxx_baseline = spectrogram(lfp_data[time_segment[0]:time_segment[1]], fs)
f = f[np.where(f<=100)] #select only frequencies below 100Hz
Sxx_baseline = Sxx_baseline[np.where(f<=100)[0],:] #select only frequencies below 100Hz

Sxx_normalized = np.empty(Sxx_evoked.shape)
for f_i in range(0, Sxx_evoked.shape[0]):
    Sxx_normalized[f_i, :] = (Sxx_evoked[f_i, :] - np.mean(Sxx_baseline[f_i, :])) /
np.std(Sxx_baseline[f_i, :])
plt.figure(figsize=(10,6))
plt.imshow(Sxx_normalized, extent=[relative_times[0], relative_times[-1], f[0], f[-1]],
aspect = 'auto', origin = 'lower', cmap='jet')
plt.xlabel('Time (ms)')
plt.ylabel('Frequency (Hz)')
```

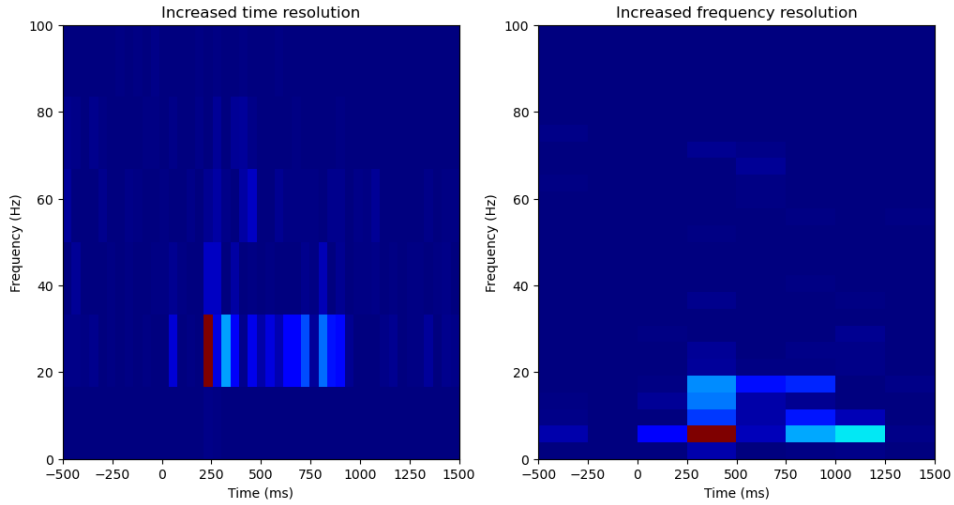
```
Text(0, 0.5, 'Frequency (Hz)')
```



The output here is clearly very blocky, due to the windowing procedure. There's also a tradeoff between time resolution and frequency resolution we can explore. Here, the bigger the time window used, the finer the frequency resolution of each power spectrum, and therefore the spectrogram as a whole. However, bigger time windows decrease time resolution as well, so we can optimize the time resolution by choosing to use smaller time windows. In scipy's spectrogram function, the length of the time segments are set by the nperseg argument, indicating the number of samples per segment. Let's try optimizing for time or frequency resolution.

```
plt.figure(figsize=(12, 6))
plt.subplot(1,2,1)
time_segment = [int(evt*fs)-500, int(evt*fs)+1500] #start and end event times of interest,
converted to ms
f, t, Sxx = spectrogram(lfp_data[time_segment[0]:time_segment[1]], fs, nperseg=50)
f = f[np.where(f<=100)] #select only frequencies below 100Hz
Sxx = Sxx[np.where(f<=100)[0],:] #select only frequencies below 100Hz
plt.imshow(Sxx, extent=[relative_times[0], relative_times[-1], f[0], f[-1]], aspect =
'auto', origin = 'lower', cmap='jet')
plt.xlabel('Time (ms)')
plt.ylabel('Frequency (Hz)')
plt.title('Increased time resolution')

plt.subplot(1,2,2)
f, t, Sxx = spectrogram(lfp_data[time_segment[0]:time_segment[1]], fs, nperseg=250)
f = f[np.where(f<=100)] #select only frequencies below 100Hz
Sxx = Sxx[np.where(f<=100)[0],:] #select only frequencies below 100Hz
plt.imshow(Sxx, extent=[relative_times[0], relative_times[-1], f[0], f[-1]], aspect =
'auto', origin = 'lower', cmap='jet')
plt.xlabel('Time (ms)')
plt.ylabel('Frequency (Hz)')
plt.title('Increased frequency resolution')
```

```
Text(0.5, 1.0, 'Increased frequency resolution')
```



Increasing the frequency resolution, we can see power is increasing in two distinct lower frequency bands, one around 4Hz and another around 20Hz.

Overall, a window-based spectrogram is a straightforward way to extract oscillatory power over both time and frequency. However, the outputs here are very limited in both time and frequency resolution. When analyzing oscillations over long (>10s) timescales, this is not a problem, and the results will look much better than these. However, we want to analyze oscillation power over a small time window (~1s), and so currently our time and frequency resolution is poor. To better analyze oscillation power in the time and frequency domains for this short time period, a morlet wavelet based analysis is more effective.

Morlet wavelets

An alternative to a windowed fourier analysis is convolution with morlet (or Gabort) wavelets. These wavelets are themselves sinewaves weighted by a gaussian. This weighting provides temporal specificity, and convolving these wavelets with the signal allows for the measurement of power over time. To analyze power from different frequencies, wavelets are created from sinewaves of different frequencies. Spectrograms produced from morelet wavelet analysis are not “blocky” like the fourier analysis, as the gaussian weighting of the wavelets results in a gaussian spread of power over both time and frequency.

Fewer packages support morlet wavelet convolution. MNE is an has a great function for this we’ll use here. The package is a great resource for analyzing LFP data in python, though it is designed for EEG+MEG data.

First, let’s set up or data for mne. We need to create an info structure, and format the data into an epochs array.

```
import mne
```

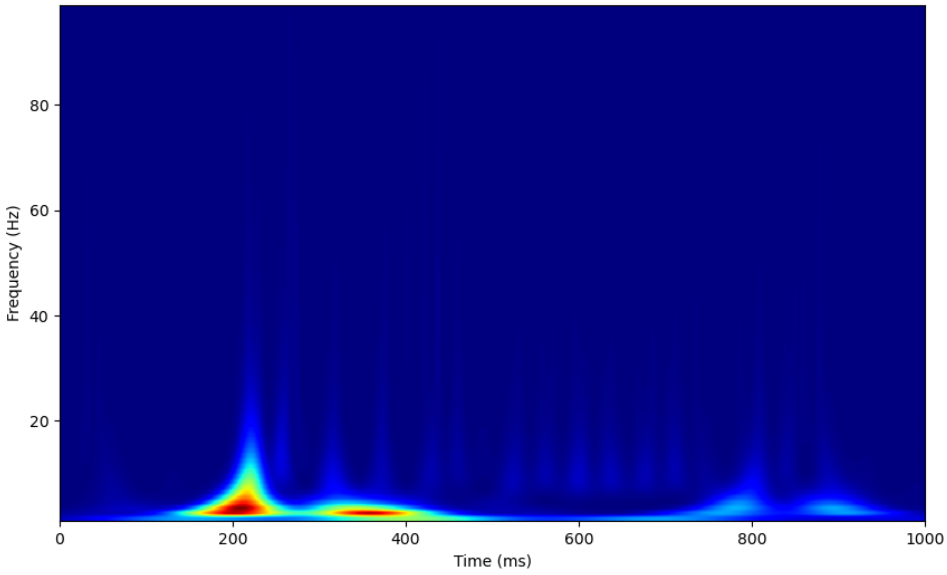
```
info = mne.create_info(ch_names=list(map(str, np.arange(1, 2, 1))), sfreq=fs, ch_types=
['eeg'])
epoch = np.empty((1, 1, len(np.arange(time_segment[0], time_segment[1],1))))
epoch[0,0,:] = lfp_data[time_segment[0]:time_segment[1]] #Format data into [epochs,
channels, samples] format
epoch = mne.EpochsArray(epoch, info, verbose=False)
```

When we run the morlet time-frequency analysis, we’ll need to set a the parameter n_cycles. Like fourier window analyses, morelet wavelet analyses have inputs that result in a trade-off between time and frequency resolution, namely, the length of the wavelet. Longer wavelets are less precise in time, but more precise in frequency. In MNE, the length of the wavelet is determined by the n_cycles parameter,

which indicates the number of cycles in the wavelets (with frequency held constant, such that more cycles results in a longer wavelet). This can be set globally for all wavelets, resulting in the same frequency specificity across wavelets of different frequencies. Let's try this first.

```
freqs = np.arange(1,100)
tf_pow = mne.time_frequency.tfr_morlet(epoch, freqs=freqs, n_cycles=1, return_itc=False,
average=False, verbose=False)
tf_pow.data = np.squeeze(tf_pow.data)
plt.figure(figsize=(10,6))
plt.imshow(tf_pow.data, extent=[relative_times[0], relative_times[1], tf_pow.freqs[0],
tf_pow.freqs[-1]],
           aspect = 'auto', origin = 'lower', cmap='jet')
plt.xlim([0, 1000])
plt.xlabel('Time (ms)')
plt.ylabel('Frequency (Hz)')
```

```
Text(0, 0.5, 'Frequency (Hz)')
```

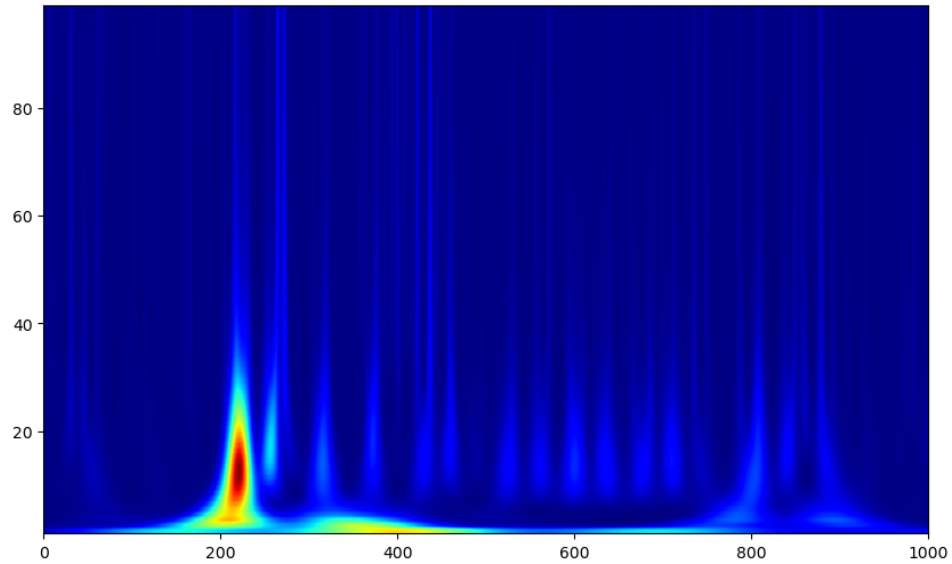


Again, we can normalize by a baseline period. MNE has an apply_baseline function on time_frequency objects we can use to here, we just need to compute the spectrogram over both the baseline and evoked periods to give the function all the data it needs.

```
relative_times = [-2000,1500]
time_segment = [int(evt*fs)+relative_times[0], int(evt*fs)+relative_times[1]] #start and
end event times of interest, converted to ms
epoch = np.empty((1, 1, len(np.arange(time_segment[0], time_segment[1],1))))
epoch[0,0,:] = lfp_data[time_segment[0]:time_segment[1]] #Format data into [epochs,
channels, samples] format
epoch = mne.EpochsArray(epoch, info, verbose=False)
freqs = np.arange(1,100)
tf_pow = mne.time_frequency.tfr_morlet(epoch, freqs=freqs, n_cycles=1, return_itc=False,
average=False, verbose=False)
tf_pow.apply_baseline(mode='zscore', baseline=(.2, 1.800)) #add a little padding for the
baseline
tf_pow.data = np.squeeze(tf_pow.data)
plt.figure(figsize=(10,6))
plt.imshow(tf_pow.data, extent=[relative_times[0], relative_times[1], tf_pow.freqs[0],
tf_pow.freqs[-1]],
           aspect = 'auto', origin = 'lower', cmap='jet')
plt.xlim([0, 1000])
```

```
Applying baseline correction (mode: zscore)
```

```
(0.0, 1000.0)
```

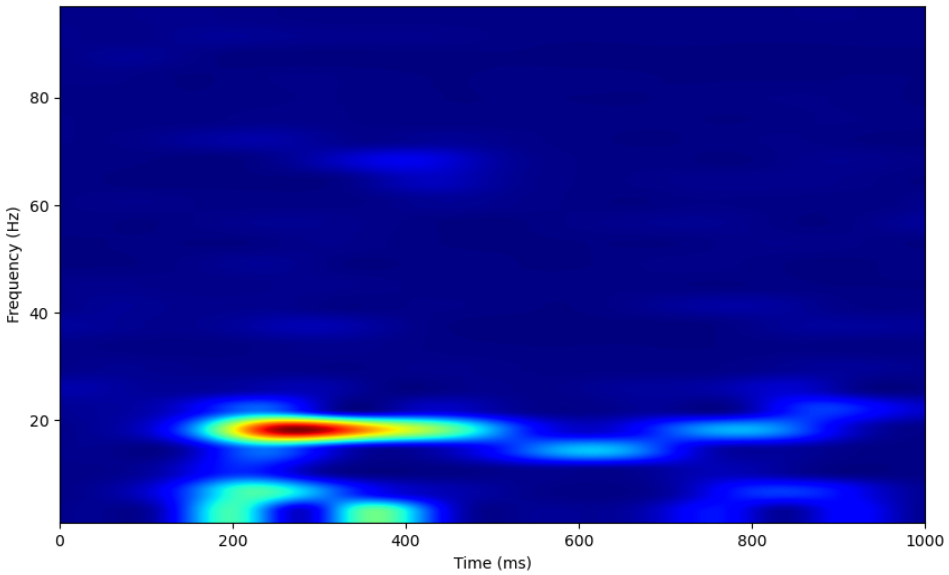


This looks better, but this spectrogram still appears smeared over time at low frequencies while varying sharply at high frequencies. This is because when we set n_cycles globally, the time specificity of each wavelet will vary, as one cycle of a 2Hz oscillation takes much more time than one cycle of a 20Hz oscillation. Therefore, weighting the number of cycles of wavelets by their frequencies is often preferable.

```
relative_times = [-2000,1500]
time_segment = [int(evt*fs)+relative_times[0], int(evt*fs)+relative_times[1]] #start and
end event times of interest, converted to ms
epoch = np.empty((1, 1, len(np.arange(time_segment[0], time_segment[1],1))))
epoch[0,0,:] = lfp_data[time_segment[0]:time_segment[1]] #Format data into [epochs,
channels, samples] format
epoch = mne.EpochsArray(epoch, info, verbose=False)
freqs = np.arange(1,100,4)
n_cycles = freqs / 2
tf_pow = mne.time_frequency.tfr_morlet(epoch, freqs=freqs, n_cycles=n_cycles,
return_itc=False, average=False, verbose=False)
tf_pow.apply_baseline(mode='zscore', baseline=(.2, 1.800)) #add a little padding for the
baseline
tf_pow.data = np.squeeze(tf_pow.data)
plt.figure(figsize=(10,6))
plt.imshow(tf_pow.data, extent=[relative_times[0], relative_times[1], tf_pow.freqs[0],
tf_pow.freqs[-1]],
           aspect = 'auto', origin = 'lower', cmap='jet')
plt.xlim([0, 1000])
plt.xlabel('Time (ms)')
plt.ylabel('Frequency (Hz)')
```

```
Applying baseline correction (mode: zscore)
```

```
Text(0, 0.5, 'Frequency (Hz)')
```



Now, we can see a clear oscillation at 20Hz, and a weak oscillation around 70Hz, around 400ms post odor presentation.

Each of these analyses measures the power of sinusoidal oscillations. However, neural oscillations are often non-sinusoidal, and this can lead to some erroneous outputs. For example, a non-sinusoidal oscillation will often lead to spurious peaks in power at harmonic frequencies of the actual oscillation frequency. This occurs when the fourier analysis decomposes the non-sinusoidal oscillation into a sinusoidal oscillation at its true frequency, and resulting in leftover signal that can be composed from harmonics of that frequency [2]

References

1. P. Welch, “The use of the fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms”, IEEE Trans. Audio Electroacoust. vol. 15, pp. 70-73, 1967.
2. Donoghue, T., Schaworonkow, N., & Voytek, | Bradley. (2021). Methodological considerations for studying neural oscillations. https://doi.org/10.1111/ejn.15361

Coherence and Directionality

This notebook will illustrate how to calculate the coherence between two signals, and then the directionality of information flow between two areas. Specifically, this notebook shows how to quantify:

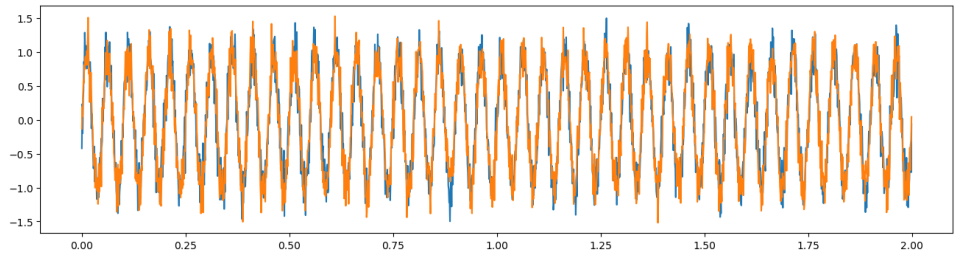
1. Coherence + phase offset in the frequecy domain
2. Coherence + phase offset in the time-frequency domains
3. Amplitude correlation
4. Phase slope index

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal
import mne_connectivity
from mne import create_info, EpochsArray
from scipy.signal import butter, filtfilt, hilbert
```

Let’s simulate two signals with a coherent 20Hz oscillation

```
fs = 1000
times = np.arange(0, 2, 1/fs)
freq = 20
osc = np.sin(2 * np.pi * times[:] * freq)
lfp1 = osc + np.random.normal(0, .2, size=times.shape)
lfp2 = osc + np.random.normal(0, .2, size=times.shape)
plt.figure(figsize=(16,4))
plt.plot(times, lfp1, label='lfp1')
plt.plot(times, lfp2, label='lfp2')
```

```
[<matplotlib.lines.Line2D at 0x1e28e270910>]
```



Now, let's measure the coherence. To do so, we'll first need to calculate the **cross spectral density** between the two signals. The difference between power and coherence comes down to performing an autocorrelation instead of a cross correlation. Where the autocorrelation of a signal is computed by convolving a signal with itself, a cross-correlation is computed by convolving one signal with another. The power spectral density is then the Fourier Transform of the autocorrelation, and the cross spectral density is the Fourier Transform of the cross-correlation.

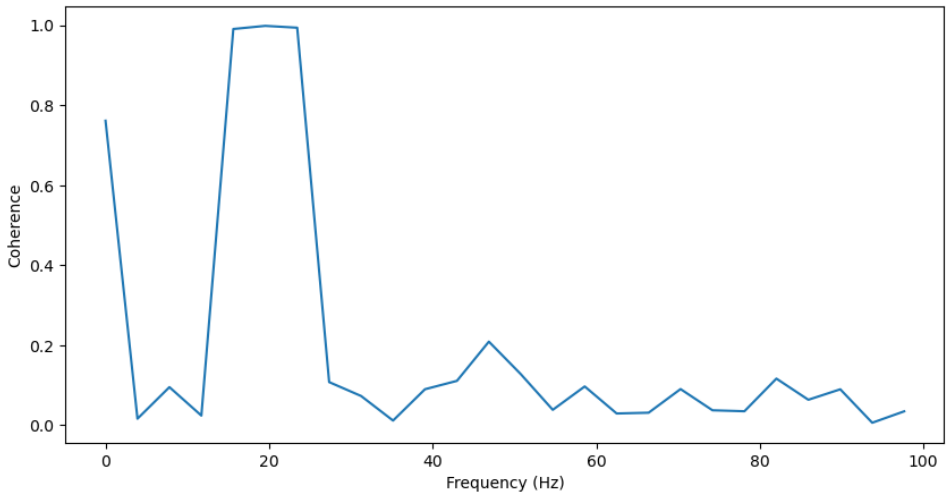
Coherence is computed as the magnitude-squared cross spectral density, normalized by the power spectrum of each signal [1]. Therefore, coherence will be between 0 (where the two signals are completely uncorrelated) and 1 (where the two signals are perfectly correlated) at each frequency.

The equation for computing coherence is therefore $Coh_{xy} = \frac{|Pxy|^2}{Pxx * Pyy}$, where Pxx is the power spectrum of x, Pyy is the power spectrum of y, and Pxy is the cross spectrum of x and y.

Therefore, the power spectrum of a signal can be thought of as the coherence of a signal with itself. As a result, we can use scipy's *csd* function both to compute the cross spectrum between x and y, and their power spectrum for computation of coherence. This function computes each spectrum using Welch's method.

```
f, Pxx = scipy.signal.csd(lfp1, lfp1, fs=fs)
f, Pyy = scipy.signal.csd(lfp2, lfp2, fs=fs)
f, Pxy = scipy.signal.csd(lfp1, lfp2, fs=fs)
Coh_xy = (np.abs(Pxy)**2) / (Pxx*Pyy)
Coh_xy = Coh_xy[np.where(f<100)]
f = f[np.where(f<100)]
plt.figure(figsize=(10,5))
plt.plot(f, Coh_xy)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Coherence')
```

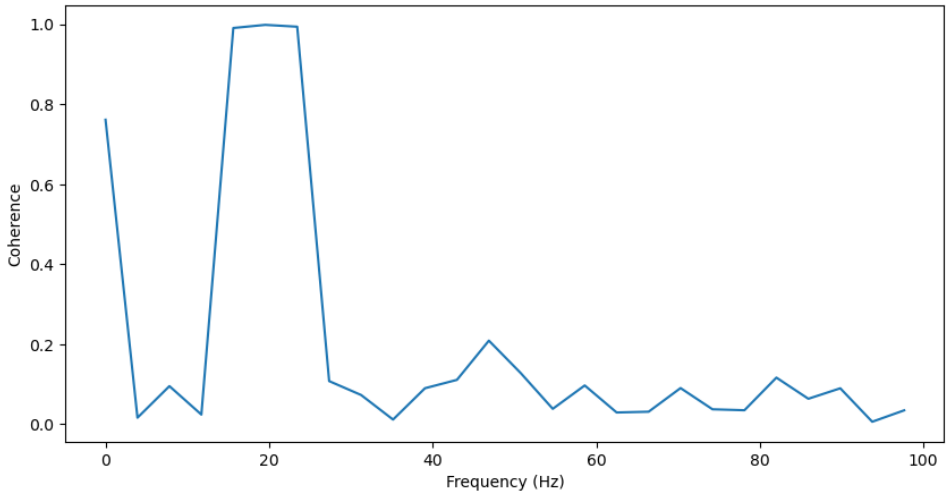
```
Text(0, 0.5, 'Coherence')
```



We can also (more conviently) use scipys *coherence* function, and obtain the same results:

```
f, coh = scipy.signal.coherence(lfp1, lfp2, fs=fs)
coh = coh[np.where(f<100)]
f = f[np.where(f<100)]
plt.figure(figsize=(10,5))
plt.plot(f, coh)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Coherence')
```

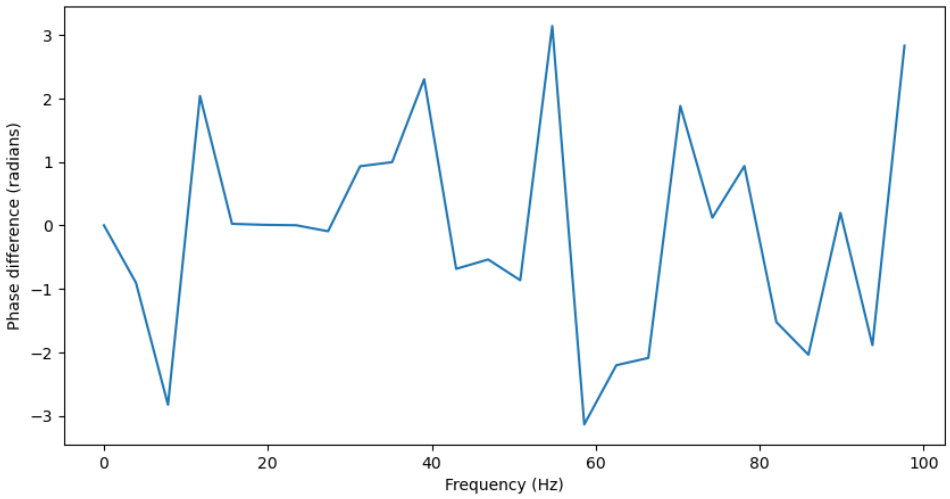
```
Text(0, 0.5, 'Coherence')
```



We can also get the phase difference between the two signals at each frequency just by taking the phase of Pxy.

```
Pxy = Pxy[:f.shape[0]]
plt.figure(figsize=(10,5))
plt.plot(f, np.angle(Pxy))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Phase difference (radians)')
```

```
Text(0, 0.5, 'Phase difference (radians)')
```



Because there is no phase lag between the two signals in the 20Hz oscillation, the this phase difference should be close to 0 at 20Hz, but essentially random at other frequencies.

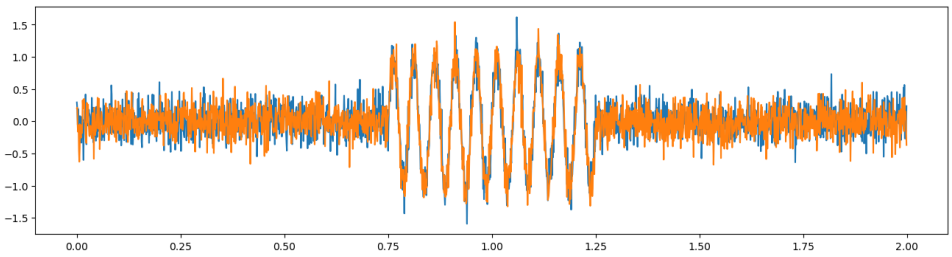
Time-frequency

Now, let's try computing coherence in the time frequency domain. This is not as straightforward as computing power in the time frequency domain. Coherence is essentially a correlation measure, and to compute a correlation, you need multiple samples from each signal. Therefore, we cannot take two signals and compute their coherence at each sample. Instead, there are two common approaches to compute coherence in the time-frequency domains. The first is to compute coherence in windows over time, or after smoothing the data over time resulting in decreased resolution over time [2]. The other is to compute the coherence over trials, rather than time, allowing for an estimation of time frequency coherence without any loss of time or frequency resolution.

Just like for time-frequency analysis of power, we can use MNE supported time-frequency decompositon via morlet wavelets to compute time-frequency coherence. First, let's simulate two signals where their coherence changes over time.

```
times = np.arange(0, 2, 1/fs)
lfp1 = np.random.normal(0, .2, size=times.shape)
lfp2 = np.random.normal(0, .2, size=times.shape)
lfp1[750:1250] += np.sin(2 * np.pi * times[:] * 20)[750:1250]
lfp2[750:1250] += np.sin(2 * np.pi * times[:] * 20)[750:1250]
plt.figure(figsize=(16,4))
plt.plot(times, lfp1, label='lfp1')
plt.plot(times, lfp2, label='lfp2')
```

```
[<matplotlib.lines.Line2D at 0x1e28f3ff220>]
```



Now let's try computing time frequency coherence over time on single trials, using MNE's spectral_connectivity_time function. We'll again need to format the data for mne (for this function, data must be formatted into an mne epochs structure).

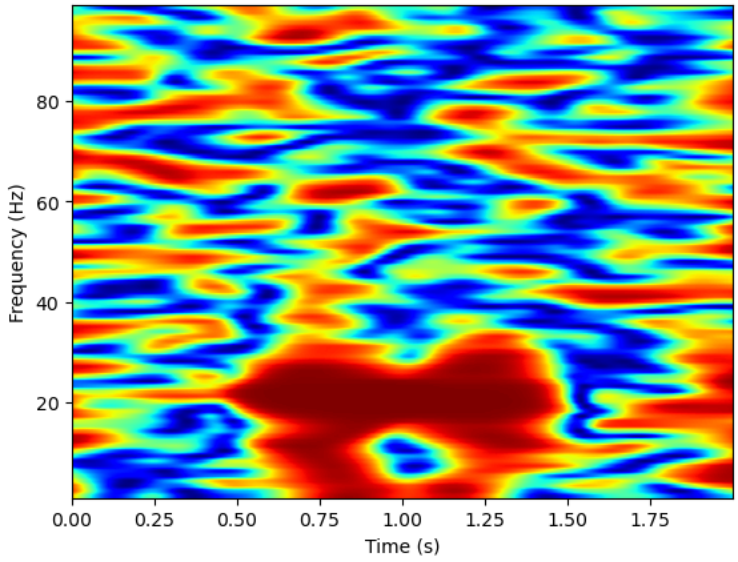
```
info = create_info(ch_names=list(map(str, np.arange(1, 3, 1))), sfreq=fs, ch_types=
['eeg','eeg'])
epochs = np.empty((1, 2, len(times)))
epochs[0,0,:] = lfp1[int(round(times[0]*fs)):int(round(times[-1]*fs))+1] #Format data into
[epochs, channels, samples] format
epochs[0,1,:] = lfp2[int(round(times[0]*fs)):int(round(times[-1]*fs))+1] #Format data into
[epochs, channels, samples] format
epochs = EpochsArray(epochs, info, verbose=False)

freqs = np.arange(1,100)
n_cycles = freqs/2
con = mne_connectivity.spectral_connectivity_time(epochs, method='coh', sfreq=int(fs),
                                                                                mode='cwt_morlet',
freqs=freqs,
                                                                                n_cycles=n_cycles,
verbose=True)
```

```
Connectivity computation...
```

```
C:\Users\Michael\AppData\Local\Temp\ipykernel_4552\2007432969.py:9: RuntimeWarning: There
were no Annotations stored in <EpochsArray |  1 events (all good), 0 - 1.999 sec, baseline
off, ~39 kB, data loaded,
 '1': 1>, so metadata was not modified.
  con = mne_connectivity.spectral_connectivity_time(epochs, method='coh', sfreq=int(fs),
```

```
#Plot
coh = con.get_data()
coh = coh[0,0,1,:,:]
plt.imshow(np.squeeze(coh), extent=[times[0], times[-1], freqs[0], freqs[-1]],
           aspect='auto', origin='lower', cmap='jet')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
# TODO: plot phase on this
```

```
Text(0, 0.5, 'Frequency (Hz)')
```



Now, we'll simulate the data over many trials, and compute the coherence over trials using MNE's spectral_connectivity_epochs function. This function takes a numpy array structure, rather than an mne epochs structure for its data input.

```
osc = np.sin(2 * np.pi * times[:] * 20)[750:1250]
trials = 10
epochs = np.empty((trials, 2, len(times)))

for trial in range(0, trials):
    lfp1 = np.random.normal(0, .2, size=times.shape)
    lfp2 = np.random.normal(0, .2, size=times.shape)
    lfp1[750:1250] += osc
    lfp2[750:1250] += osc
    epochs[trial, 0, :] = lfp1
    epochs[trial, 1, :] = lfp2
```

```
freqs = np.arange(1,100)
n_cycles = freqs/2
con = mne_connectivity.spectral_connectivity_epochs(epochs, method='coh', sfreq=int(fs),
                                                    mode='cwt_morlet', cwt_freqs=freqs,
                                                    cwt_n_cycles=n_cycles, verbose=True,
                                                    indices=(np.array([0]), np.array([1])))
```

```
Connectivity computation...
```

```
    computing connectivity for 1 connections
```

```
    using t=0.000s..1.999s for estimation (2000 points)
```

```
    frequencies: 3.0Hz..99.0Hz (97 points)
```

```
    using CWT with Morlet wavelets to estimate spectra
```

```
    the following metrics will be computed: Coherence
```

```
    computing connectivity for epoch 1
```

```
    computing connectivity for epoch 2
```

```
    computing connectivity for epoch 3
```

```
    computing connectivity for epoch 4
```

```
    computing connectivity for epoch 5
```

```
    computing connectivity for epoch 6
```

```
    computing connectivity for epoch 7
```

```
    computing connectivity for epoch 8
```

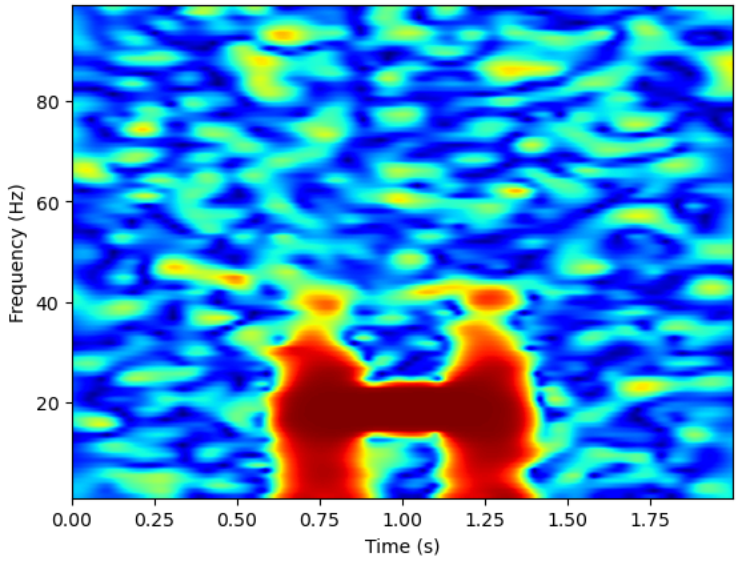
```
    computing connectivity for epoch 9
```

```
    computing connectivity for epoch 10
```

```
[Connectivity computation done]
```

```
coh = con.get_data()

plt.imshow(np.squeeze(coh), extent=[times[0], times[-1], freqs[0], freqs[-1]],
           aspect='auto', origin='lower', cmap='jet')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
```

```
Text(0, 0.5, 'Frequency (Hz)')
```



Phase

Just as for coherence in frequency space, we can get the phase difference of time-frequency coherence. To do so, we first get the complex coherence values, containing both the amplitude and phase information of the coherence, by setting the method parameter of the spectral_connectivity function to 'cohy'. We can get the amplitude of the coherence (the same metric computed originally) by taking the absolute value of this output, and the phase by taking the angle.

```
con_complex = mne_connectivity.spectral_connectivity_epochs(epochs, method='cohy',
sfreq=int(fs),
                                                            mode='cwt_morlet', cwt_freqs=freqs,
                                                            cwt_n_cycles=n_cycles, verbose=False,
                                                            indices=(np.array([0]), np.array([1])))

coh_complex = con_complex.get_data()
coh_phase = np.angle(coh_complex)
coh = np.abs(coh_complex)
```

```
Connectivity computation...
```

```
    computing connectivity for 1 connections
```

```
    using t=0.000s..1.999s for estimation (2000 points)
```

```
    frequencies: 3.0Hz..99.0Hz (97 points)
```

```
    using CWT with Morlet wavelets to estimate spectra
```

```
    the following metrics will be computed: Coherency
```

```
    computing connectivity for epoch 1
```

```
    computing connectivity for epoch 2
```

```
    computing connectivity for epoch 3
```

```
    computing connectivity for epoch 4
```

```
    computing connectivity for epoch 5
```

```
    computing connectivity for epoch 6
```

```
    computing connectivity for epoch 7
```

```
    computing connectivity for epoch 8
```

```
    computing connectivity for epoch 9
```

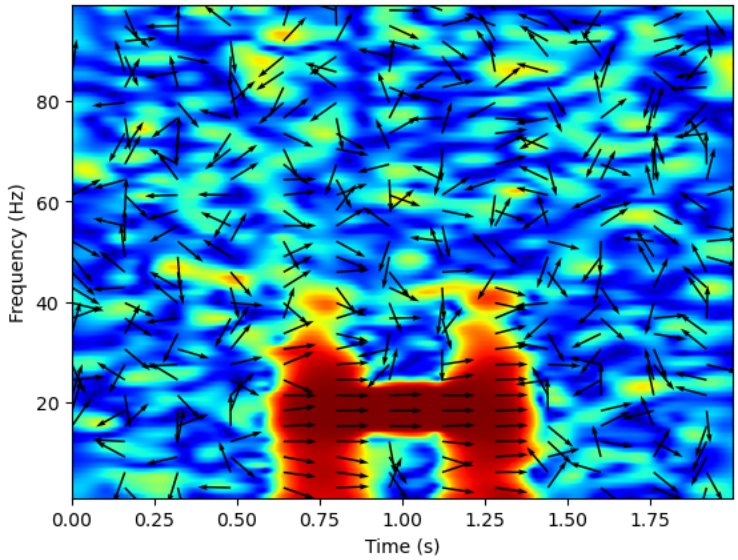
```
    computing connectivity for epoch 10
```

```
[Connectivity computation done]
```

We can plot both the amplitude and phase of the coherence on the same plot by plotting the phase information as a quiver plot. Since the coherence output contains data for every sample and every frequency, I select a subset of the phase values to plot.

```
plt.imshow(np.squeeze(coh), extent=[times[0], times[-1], freqs[0], freqs[-1]],
           aspect='auto', origin='lower', cmap='jet')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
coh_phase = np.squeeze(coh_phase)
X, Y = np.meshgrid(np.linspace(times[0], times[-1], coh_phase.shape[1]),
np.linspace(freqs[0], freqs[-1], coh_phase.shape[0]))

U = np.cos(coh_phase)
V = np.sin(coh_phase)
f_x = 160
f_y = 3
plt.quiver(X[2::f_y, ::f_x], Y[2::f_y, ::f_x], U[2::f_y, ::f_x], V[2::f_y, ::f_x],
angles='uv', scale=20)
```

```
<matplotlib.quiver.Quiver at 0x1e28fb93be0>
```



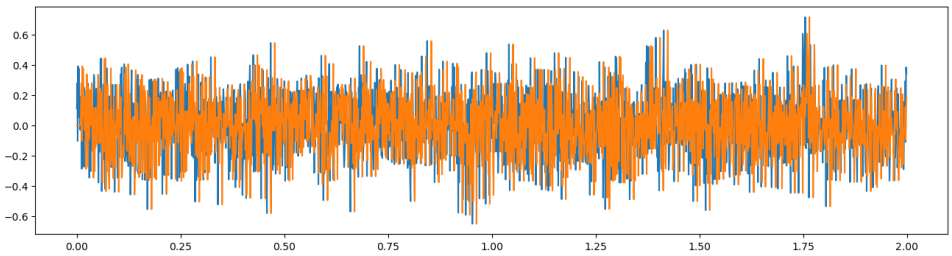
Directionality

If two signals (brain areas) are correlated, that could mean they both are affecting each other's activity, or one area is primarily driving another. Therefore, after analyzing the coherence between two signals, it is often useful to analyze the directionality of any relationship between two signals. Here, I will discuss two common methods for assessing directionality in neuroscience: **amplitude correlations** and the **phase slope index**.

First, we'll simulate data such that lfp2 lags lfp1.

```
# Simulate data
lag = 10
samples = 2000
lfp2 = np.zeros(samples)
lfp1 = np.random.normal(0, .2, size=samples)
lfp2[lag:] = lfp1[:samples-lag]

# lfp1[750:1250] += np.sin(2 * np.pi * times[:] * 20)[0:500]
# lfp2[750+lag:1250+lag] += lfp1[750:1250]

plt.figure(figsize=(16,4))
plt.plot(times, lfp1, label='lfp1')
plt.plot(times, lfp2, label='lfp2')
```

```
[<matplotlib.lines.Line2D at 0x1e28fa36070>]
```



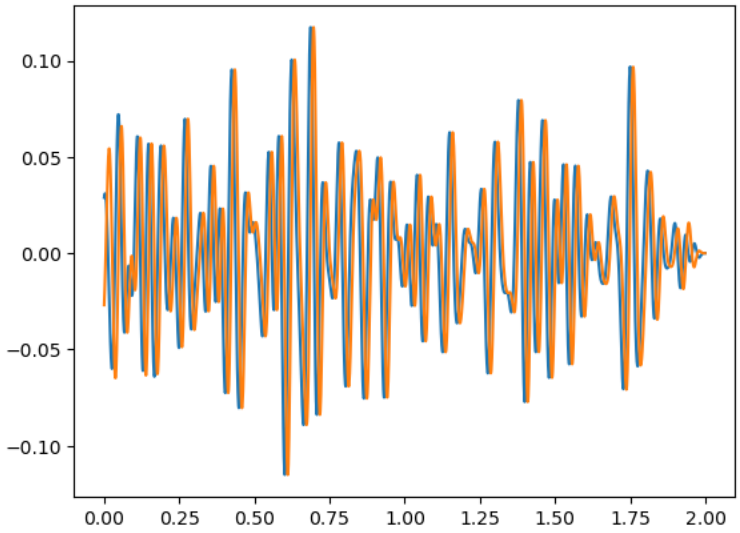
Amplitude Correlation

The amplitude correlation is a straightforward method of computing the directionality between two signals. Also referred to as a power envelope correlation [3], the method is a cross-correlation of the amplitudes in a frequency between two signals. The first step of this method is to filter each signal in the frequency band of interest, and then compute the amplitude envelope of each by taking the hilbert transform of the filtered signal. Then a cross-correlation is performed between the two amplitude

envelopes. Therefore, the amplitude correlation can indicate when the power of the oscillation is most correlated between the two signals, revealing whether an oscillation in one signal precedes or follows an oscillation in another signal.

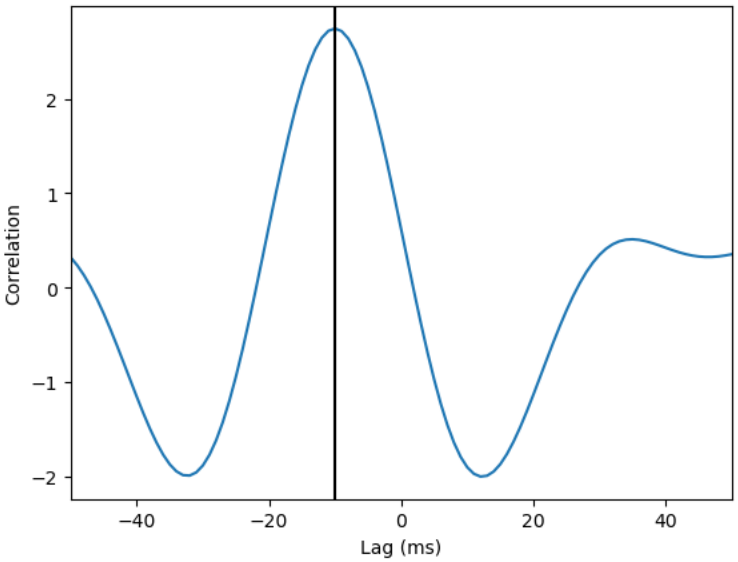
```
#Filter
order=4
freq_band = [12, 32]
b, a = butter(order, freq_band, fs=fs, btype='band')
lfp1_filt = filtfilt(b, a, lfp1, padtype=None)
lfp2_filt = filtfilt(b, a, lfp2, padtype=None)
plt.plot(times, lfp1_filt)
plt.plot(times, lfp2_filt)
```

```
[<matplotlib.lines.Line2D at 0x1e28fb48c70>]
```




```
lfp1_analytic = hilbert(lfp1_filt)
lfp2_analytic = hilbert(lfp2_filt)
xcorr = scipy.signal.correlate(lfp1_filt, lfp2_filt)
lags = scipy.signal.correlation_lags(len(lfp1_analytic), len(lfp2_analytic))
plt.plot(lags, xcorr)
plt.xlim(-50, 50)
plt.xlabel('Lag (ms)')
plt.ylabel('Correlation')
plt.axvline(lags[np.argmax(xcorr)], color='black')
print('Correlation is maximal where lag = '+str(lags[np.argmax(xcorr)])+'ms')
```

```
Correlation is maximal where lag = -10ms
```



The two signals are most correlated when the second signal is shifted backwards in time by 10ms, indicating the second signal lags the first by 10ms.

Phase Slope Index

A second method to assess the directionality of information flow is the phase slope index (PSI) [4]. Phase is circular, and thus cannot indicate directionality in itself. However, taking the phase difference over a range of frequencies, one can begin to interpret directionality. PSI assumes one brain area interacts with another with some time offset, such that the speed at which different waves travel is similar. Since the same time difference will result in larger phase differences as frequency increases, there should be a positive slope in the phase spectrum. Therefore, PSI quantifies the slope of the phase spectrum, such that a positive slope indicates the first signal leads the second, and a negative signal indicates the reverse.

```
# Compute PSI
epochs = np.empty((1, 2, len(times)))
epochs[0,0,:] = lfp1 #Format data into [epochs, channels, samples] format
epochs[0,1,:] = lfp2 #Format data into [epochs, channels, samples] format
minf = 60
maxf = 100
freqs = np.arange(minf, maxf, 2)
n_cycles = freqs/2
# con = mne_connectivity.SpectralConnectivity(epochs, n_nodes=2,
freqs=np.arange(minf,maxf,2), method='cohy', indices=[np.asarray(0), np.asarray(1)])
con = mne_connectivity.phase_slope_index(epochs, sfreq=int(fs), verbose=True, fmin=minf,
fmax=maxf, indices=(np.array([0]), np.array([1])))
```

```
Estimating phase slope index (PSI)
```

```
Connectivity computation...
```

```
    computing connectivity for 1 connections
```

```
    using t=0.000s..1.999s for estimation (2000 points)
```

```
    frequencies: 60.0Hz..100.0Hz (81 points)
```

```
    Using multitaper spectrum estimation with 7 DPSS windows
```

```
    the following metrics will be computed: Coherency
```

```
    computing connectivity for epoch 1
```

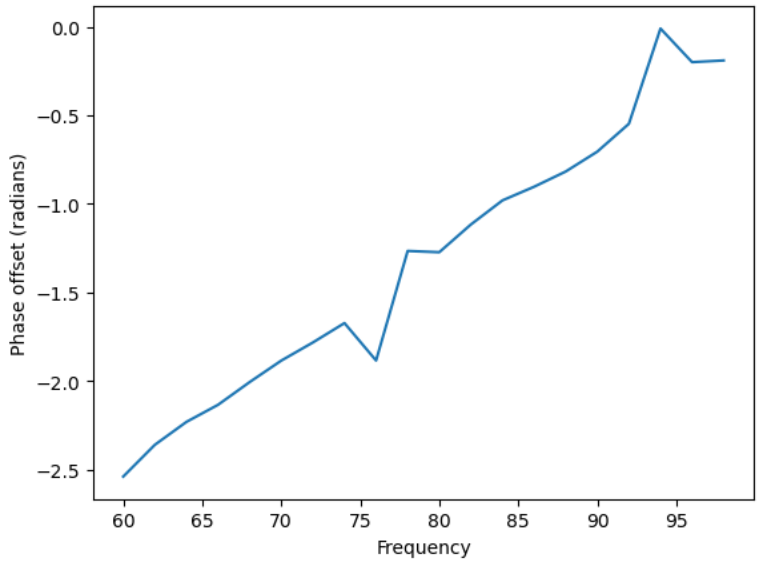
```
[Connectivity computation done]
```

```
Computing PSI from estimated Coherency: <SpectralConnectivity | freq : [60.000000,
100.000000], , nave : 1, nodes, n_estimated : 2, 1, ~68 kB>
```

```
[PSI Estimation Done]
```

```
# Plot the relationship between phase difference and frequency
con_complex = mne_connectivity.spectral_connectivity_epochs(epochs, method='cohy',
sfreq=int(fs),
                                                    mode='cwt_morlet', cwt_freqs=freqs,
                                                    cwt_n_cycles=n_cycles, verbose=False,
                                                    indices=(np.array([0]), np.array([1])))

coh_complex = con_complex.get_data()
coh_phase = np.angle(coh_complex)

#Plot the phase slope for one timepoint
plt.plot(freqs, coh_phase[0,:,1000])
plt.xlabel('Frequency')
plt.ylabel('Phase offset (radians)')
```

```
Connectivity computation...
    computing connectivity for 1 connections
    using t=0.000s..1.999s for estimation (2000 points)
    frequencies: 60.0Hz..98.0Hz (20 points)
    using CWT with Morlet wavelets to estimate spectra
    the following metrics will be computed: Coherency
    computing connectivity for epoch 1
[Connectivity computation done]
Text(0, 0.5, 'Phase offset (radians)')
```




```
psi = con.get_data()
print(psi)
```

```
[[2.49877028]]
```

The slope of the phase offset by frequency graph is positive, leading to a positive PSI. This suggests lfp1 leads lfp2, as we know to be correct.

References

1. Stoica, Petre, and Randolph Moses, “Spectral Analysis of Signals” Prentice Hall, 2005
2. Sandberg, J., & Hansson, M. (2006). Coherence estimation between EEG signals using multiple window time-frequency analysis compared to Gaussian kernels. IEEE.
3. Hipp, J. F., Hawellek, D. J., Corbetta, M., Siegel, M., & Engel, A. K. (2012). Large-scale cortical correlation structure of spontaneous oscillatory activity. Nature Neuroscience 2012 15:6, 15(6), 884–890. https://doi.org/10.1038/nn.3101
4. Nolte, G., Ziehe, A., Nikulin, V. V., Schlögl, A., Krämer, N., Brismar, T., & Müller, K. R. (2008). Robustly estimating the flow direction of information in complex physical systems. Physical Review Letters, 100(23). https://doi.org/10.1103/PHYSREVLETT.100.234101

Spike-LFP Coupling

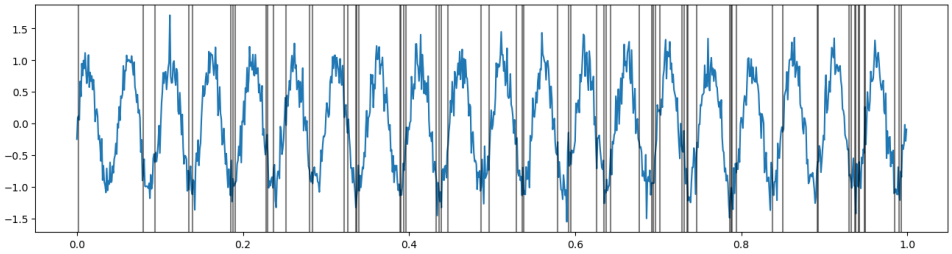
Neural oscillations affect the spike timing of neurons. Identifying how neurons fire in relation to oscillations can help uncover which neurons participate in an oscillation, and and to what degree. This notebook will demonstrate how to quantify how the spike timing of neurons relates to underlying oscillations. Specifically, this notebook shows how to:

1. Filter data into a frequency band
2. Extract phase information
3. Visualize spike-lfp coupling
4. Quantify spike-lfp coupling

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt, hilbert
import math
import itertools
import random
import time
```

Let's start by simulating an oscillation, and a neuron that tends to fire at the trough of the oscillation. We'll plot the spikes as vertical lines over the lfp.

```
#Simulate LFP
fs = 1000
times = np.arange(0, 1, 1/fs)
freq = 20
lfp = np.sin(2 * np.pi * times[:] * freq)
noise = np.random.normal(0, .2, size=times.shape)
lfp += noise
plt.figure(figsize=(16, 4))
plt.plot(times, lfp)
#Simulate neuron
spike_times = []
lfp_min0 = lfp-np.min(lfp)+.001
for i in range(0,len(times)):
    probability = (2 - lfp_min0[i])/10
    if probability > 0:
        sample = np.random.choice([0,1],p=[1-probability, probability])
        if sample==1:
            spike_times.append(times[i])
for spike in spike_times:
    plt.axvline(spike, color='black', alpha=.5)
```



We'll want to look at how the neuron fires with respect to some particular oscillation, or frequency band. To do so, we need to filter the lfp signal into a range of interest. Let's start by filtering in the beta (12-32Hz) band.

Filtering and Phase

Filtering is used to emphasize or demphasize activity in particular frequencies in time-series. Common applications of filters to neural electrophysiological data are to remove 60Hz line noise, remove low frequency activity for spike detection / remove high frequency activity for LFP analyses, and to isolate specific oscillations.

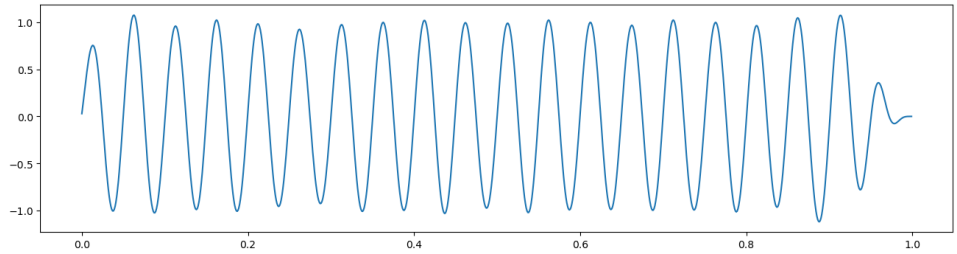
There are many commonly used filter designs, with customizable parameters. The goal of each filter is to maximize the attennuation of unwanted frequencies, which can be best achieved using filters with a narrow transition band in frequency space. However, these "sharp" filters produce stronger signal distortions and ringing artifacts. Therefore, choices in filter design often come down to this trade-off

[1]. Additionally, higher filter orders effectively decrease the time resolution while increasing the frequency resolution. Here, we'll use a butterworth filter, which is maximally uniform in how it affects the magnitude of activity in the filtered frequency range. Additionally, we'll use an order of 4. Feel free to try other filter designs (cheby2, bessel) or orders to see their affects.

Filters are often applied through convolution over the signal. This can distort phase estimation, as convolving in a particular direction will result in a phase shift. To eliminate this shift, filters can be convolved once forwards and then backwards, which we will do with scipy's filtfilt function.

```
order=4
freq_band = [12, 32]
b, a = butter(order, freq_band, fs=fs, btype='band')
filt_data = filtfilt(b, a, lfp, padtype=None)
plt.figure(figsize=(16,4))
plt.plot(times, filt_data)
```

```
[<matplotlib.lines.Line2D at 0x1e064233400>]
```

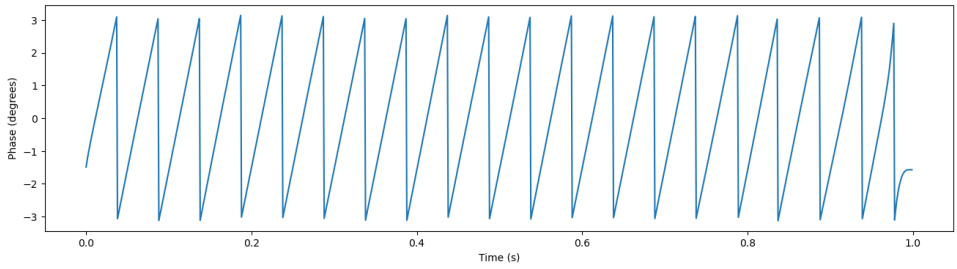


As you can see, we've removed a lot of the noise and cleanly uncovered the underlying oscillation.

Now we have extracted our oscillation, but we want to see whether the neuron's spike timing is affected by this oscillation. One useful way to investigate this question is to look at whether the neuron tends to fire at some particular phase of the oscillation more than others. We therefore start by extracting the phase information for this oscillation. To get this, you use the hilbert transform to produce the analytic signal. This analytic signal is complex valued and contains both the instantaneous amplitude and phase information for our signal. The phase can be easily extracted by calling np.angle() function on the analytic signal.

```
analytic_signal = hilbert(filt_data)
phase = np.angle(analytic_signal, deg=False)
plt.figure(figsize=(16,4))
plt.plot(times, phase)
plt.xlabel('Time (s)')
plt.ylabel('Phase (degrees)')
```

```
Text(0, 0.5, 'Phase (degrees)')
```



Quantify Spike LFP-Coupling

Now let's try to see how the spike times of the neuron align to this beta oscillation. A good way to start is to create a histogram of these spike times by binning over their oscillation phase.

First, let's collect the phases when the neuron spiked.

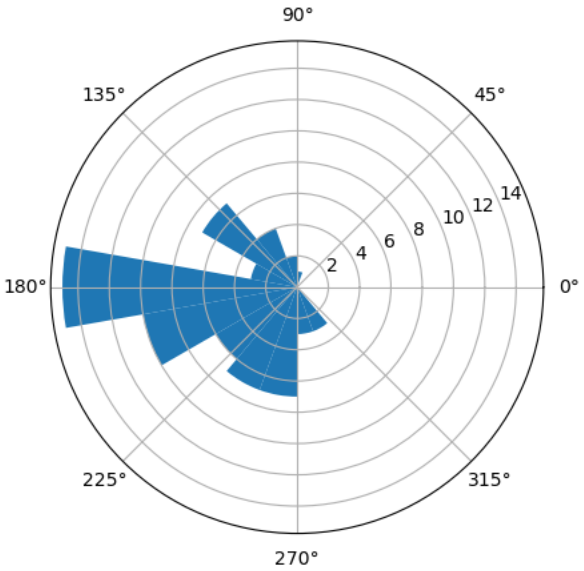
```
spike_phases = []
for spike_time in spike_times:
    spike_phases.append(phase[int(round(spike_time*fs))])
```

To create the histogram, we then bin these spike phases using numpy's digitize function, and then count up how many phases we have in each phase bin.

```
n_bins = 18
phase_bins = np.linspace(-np.pi, np.pi, n_bins+1)
binned_phases = np.digitize(spike_phases, phase_bins, n_bins+1)
spike_phase_hist = np.zeros(n_bins+1)
for bin in range(0, n_bins + 1):
    spike_phase_hist[bin] = np.sum(binned_phases == bin)
print(spike_phase_hist)

#Plot
ax = plt.subplot(1,1,1, polar=True)
plt.bar(phase_bins, spike_phase_hist, width=phase_bins[1] - phase_bins[0], bottom=0.0)
```

```
[ 0. 10.  6.  7.  7.  3.  3.  0.  0.  0.  0.  0.  0.  1.  2.  4.  7.  3.
 15.]
```

```
<BarContainer object of 19 artists>
```



We can now see spikes are more likely to fire around 180 degrees phase, or the trough, of the oscillation. Let's try to quantify the extent of this spike-phase coupling. To do so, I'll use mean vector length, and pairwise phase coherence.

Mean Vector length

A common way to measure spike phase locking is to take the length of the mean vector of the spike-phase distribution [2]. If spikes are uniformally distributed with respect to phase, the mean of this distribution will be a point close to the center of the circle, and its radius will be close to 0. On the other hand, if spikes are clustered around some particular phase, the mean vector will point towards that phase and thus will have a radius closer to 1.

To calculate the mean vector length (mvl) of our distribution, we first need to map the spike phase distribution into polar coordinates and then take their mean. The mvl is then the distance of this point from (0,0).

```
def MVL(spike_phases):
    a_cos = map(lambda x: math.cos(x), spike_phases)
    a_sin = map(lambda x: math.sin(x), spike_phases)
    a_cos, a_sin = np.fromiter(a_cos, dtype=float), np.fromiter(a_sin, dtype=float)
    uv_x = sum(a_cos)/len(a_cos)
    uv_y = sum(a_sin)/len(a_sin)
    uv_radius = np.sqrt((uv_x*uv_x) + (uv_y*uv_y))
    p_value = np.round(np.exp(-1 * len(spike_phases) * (uv_radius ** 2)), 6)
    return uv_radius, p_value

vec_length, pval = MVL(spike_phases)
print('Mean vector length: '+ str(round(vec_length,3))+', p value: '+str(pval))
```

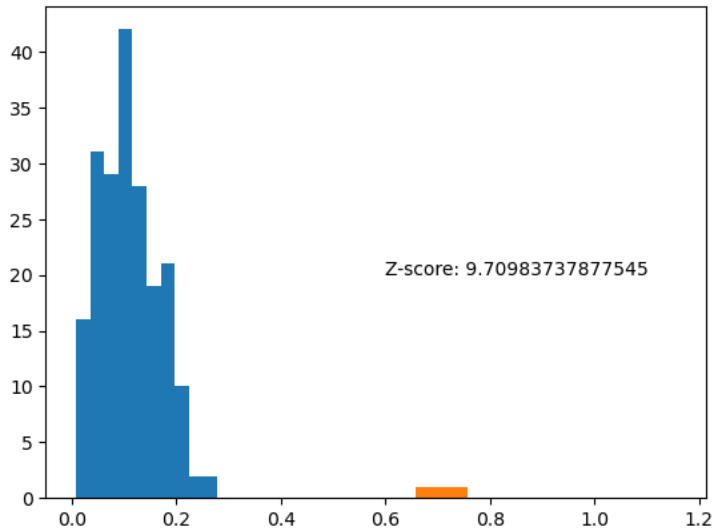
```
Mean vector length: 0.657, p value: 0.0
```

We get a p value of 0, indicating phase locking is highly significant. However, this measure is highly dependent on the number of spikes in our sample [3], as small samples will lead to a mean centered away from (0,0) even if spike phases are uniformally distributed.

One way to get around this is to repeatedly estimate the mvl on a surrogate spike phase distribution with the same number of samples as our actual sample, and estimate the difference between our real result and these surrogate estimates.

```
n_surrogates = 200
sample_size = len(spike_phases)
mvl_surrogates = [None] * n_surrogates
for surr in range(0, n_surrogates):
    spike_phases_surr = random.choices(phase, k=sample_size) #see note
    mvl_surrogates[surr], p = MVL(spike_phases_surr)

z_score = (vec_length - np.mean(mvl_surrogates)) / np.std(mvl_surrogates)
plt.hist(mvl_surrogates)
plt.hist(vec_length)
plt.text(.6, 20, 'Z-score: '+ str(z_score))
```

```
Text(0.6, 20, 'Z-score: 9.70983737877545')
```



The real mvl is clearly significantly higher than these surrogates, indicating spike-phase coupling is significant.

Note: We could have randomly sampled from any number between -np.pi and np.pi, but instead I sampled from the actual observed phase values. This controls for a potential bias of spike-phase locking estimation, where not all phase values are observed in the oscillation itself. This occurs in smaller samples, where the length of the sample is less than a full cycle. If not all phases are observed, than the spike-phase distribution will be artificially non-uniform, and significant when compared to a surrogate distribution containing all possible phase values.

Pairwise Phase Consistency

Alternatively, another measure of spike-phase locking, pairwise phase consistency (ppc), is unbiased by the number of samples [3]. The measure computes the cosine of the angular distance between each pair of phases, resulting in an output of 1 for pairs of the same phase and -1 for antiphase pairs. After averaging across pairs, the measure will approach 0 for a uniform phase distribution and 1 for a perfectly aligned phase distribution.

```
def PPC(spike_phases):
    sp_complex = map(lambda x: [math.cos(x), math.sin(x)], spike_phases)
    all_com = list(itertools.combinations(sp_complex , 2))
    dp_array = np.empty(int(len(spike_phases) * (len(spike_phases) - 1) / 2))
    for d, combination in enumerate(all_com):
        dp = np.dot(combination[0], combination[1])
        dp_array[d] = dp
    dp_sum = np.sum(dp_array)
    ppc = dp_sum / len(dp_array)
    return ppc

ppc_val = PPC(spike_phases)
print('PPC: ' + str(ppc_val))
```

```
PPC: 0.4230545209504523
```

The only disadvantage of using ppc over mvl is ppc, by computing the dot product between each possible phase pair, takes longer. Let's illustrate that by measuring the time each function takes for a sample with 1000 phases.

```
spike_phases = [np.pi for x in range(0,1000)]
start = time.process_time()
MVL(spike_phases)
stop = time.process_time()
print('Mean Vector Length process time: ' + str(stop-start))
start = time.process_time()
PPC(spike_phases)
stop = time.process_time()
print('Pairwise Phase Consistency process time: ' + str(stop-start))
```

```
Mean Vector Length process time: 0.0
```

```
Pairwise Phase Consistency process time: 1.453125
```

Clearly the PPC take significantly longer to compute. Though this time is still tolerable, or if you were to run ppc on many surrogate distributions (to capture the bias of the measure due to an incomplete phase distribution, for example), this time can add up. Therefore, when comparing with surrogate data, the mvl is more often used.

References

1. Widmann, A., Schröger, E., & Maess, B. (2015). Digital filter design for electrophysiological data – a practical approach. Journal of Neuroscience Methods, 250, 34–46. https://doi.org/10.1016/J.JNEUMETH.2014.08.002
2. Canolty RT, Edwards E, Dalal SS, Soltani M, Nagarajan SS, Kirsch HE, et al. High Gamma Power Is Phase-Locked to Theta Oscillations in Human Neocortex. Science (80-) [Internet]. 2006 Sep 15 [cited 2019 Aug 7];313(5793):1626–8. Available from: http://www.sciencemag.org/cgi/doi/10.1126/science.1128115
3. Vinck M, van Wingerden M, Womelsdorf T, Fries P, Pennartz CMA. The pairwise phase consistency: A bias-free measure of rhythmic neuronal synchronization. Neuroimage. 2010 May 15;51(1):112–22.

Spike Train Analyses

This tutorial will introduce analyses of neural spiking data, including identifying rhythmic activity in a neuron, and examining stimulus encoding by neural spike counts. Specifically, we will perform:

1. Raster plot
2. Spike rate estimation using kernals
3. Autocorrelation
4. Power analyses of spike rate data (PSD, time-frequency power)
5. PCA visualization

```
import numpy as np
import matplotlib.pyplot as plt
import elephant
from neo.core import AnalogSignal
import quantities as pq
import mne
from scipy.signal import welch
from sklearn.decomposition import PCA
```

First, we will simulate a neural spike train, an array of spike times for a neuron. For this and later analyses, we will use elephant [1]. Though elephant supports analyses of spike trains and LFP data, the package is particularly helpful for spike train analyses. For input, elephant often requires input data to be formatted using the quantity and neo packages: quantity allows the unit of measurement (seconds, milliseconds, Hz) to be tied to a numpy array, and neo supports electrophysiological signal data objects, allowing data to be packaged with relevant information (such as the recorded time range of a spike train).

Here, we'll simulate a spike train using elephant's spike_train_generation module. We will use the inhomogeneous_poisson_process function, allowing us to specify a firing rate that fluctuates rhythmically at 20Hz.

```
# Simulate a neuron with an oscillating firing rate

fs = 1000
times = np.arange(0, 1, 1/fs)
freq = 20
mean_rate = 60
osc = np.sin(2 * np.pi * times[:] * freq)
osc_rate = mean_rate * osc
osc_rate += mean_rate #make sure rate is always positive
sim_spike_rate = AnalogSignal(np.expand_dims(osc_rate, 1), units='Hz',
sampling_rate=1000*pq.Hz)
spiketrain =
elephant.spike_train_generation.inhomogeneous_poisson_process(rate=sim_spike_rate,
refractory_period=3*pq.ms)
```

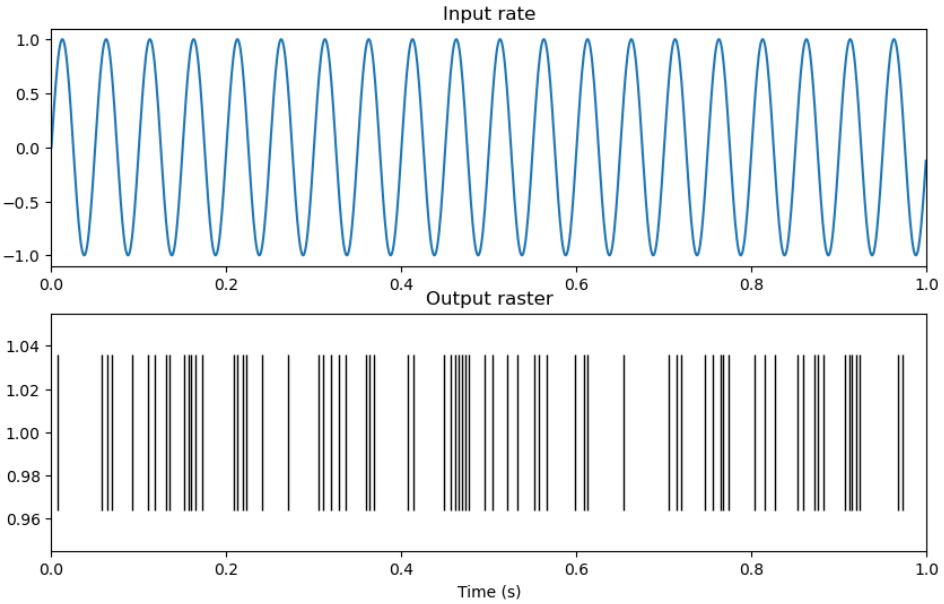
Raster plot

The *spiketrain* output now contains an array of spike-times, which we can visualize using a raster plot. A raster plot visualizes spike times as vertical lines.

```
plt.figure()
spikes = np.squeeze(spiketrain)
plt.figure(figsize=(10,6))
plt.subplot(2,1,1)
plt.plot(times, osc)
plt.title('Input rate')
plt.xlim([0,1])
# plt.tick_params(bottom = False, labelbottom = False)
plt.subplot(2,1,2)
plt.plot(spikes, np.ones_like(spikes), '|', markersize=100, color='black')
plt.xlim([0,1])
plt.title('Output raster')
plt.xlabel('Time (s)')
```

```
Text(0.5, 0, 'Time (s)')
```

```
<Figure size 640x480 with 0 Axes>
```

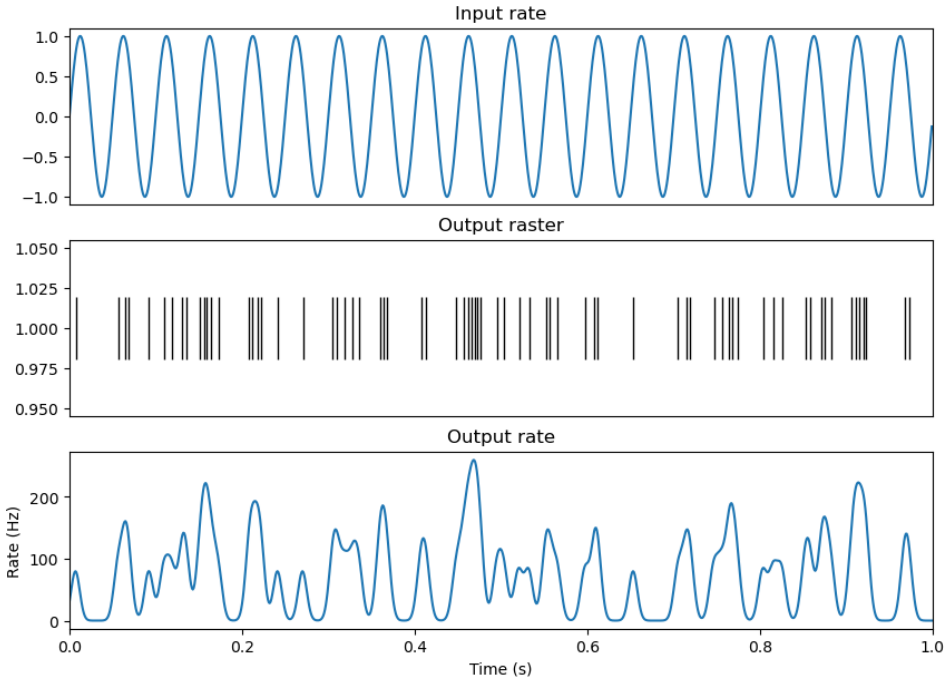


Spike rate estimation using kernel density estimation

It is often useful to estimate how the spike rate of a neuron changes over time. A classical and effective method is kernel density estimation, where a kernel function is convolved with a spike train [2]. Common kernel functions include a gaussian kernel, which facilitates a smooth estimate of spike rate over time, or an exponential kernel, which can be used to prevent the smoothing of spike rate estimation backwards in time. Additionally, the user must decide the width of the kernel. Wider kernels smooth the spike rate estimation over longer periods of time (and thus risk smoothing over important changes in spike rate), while narrow kernels can produce noisy fluctuations in firing rate estimation. Here, I've selected a guassian kernel with a width (or standard deviation in time) of 5ms, where we can observe firing rate increase and decrease over the 20Hz (50ms cycle) rhythm.

```
# Apply a kernel to extract firing rate from spike train

kernel_size = 5*pq.ms
spike_sr = 1000
real_spike_rate = elephant.statistics.instantaneous_rate(spiketrain, sampling_period=1 /
spike_sr * pq.s,

kernel=elephant.kernels.GaussianKernel(sigma=kernel_size))
plt.figure(figsize=(10, 7))
plt.subplot(3,1,1)
plt.plot(times, osc)
plt.title('Input rate')
plt.xlim([0,1])
plt.tick_params(bottom = False, labelbottom = False)
plt.subplot(3,1,2)
plt.plot(spikes, np.ones_like(spikes), '|', markersize=40, color='black')
plt.title('Output raster')
plt.tick_params(bottom = False, labelbottom = False)
plt.xlim([0,1])
plt.subplot(3,1,3)
plt.plot(times, np.squeeze(np.asarray(real_spike_rate)))
plt.xlim([0,1])
plt.title('Output rate')
plt.xlabel('Time (s)')
plt.ylabel('Rate (Hz)')
```

```
Text(0, 0.5, 'Rate (Hz)')
```



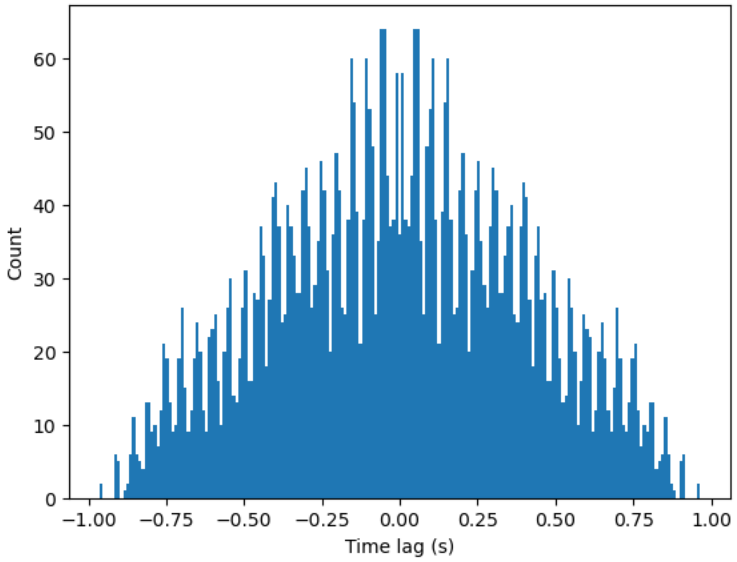
Rhythmicity

Autocorrelation

Performing an autocorrelation on this spiketrain will allow us to capture the rhythmicity. Generally, an autocorrelation of a spike train computes the difference between the spike time of each spike with all other spike times in the spike train. If the neuron fires rhythmically, this will be reflected in the autocorrelation as an increased number of spikes at multiples of some time difference relative to 0.

```
# Compute autocorrelation of neuron

# spikes = [t.item() for t in spikes] #convert spiketrain into a list of spike times

binsize = .005
histo_bins = int((times[-1]-times[0])/binsize)
spike_diffs = []
for i, spike1 in enumerate(spikes):
    for j, spike2 in enumerate(spikes):
            if not spike1==spike2:
                spike_diffs.append(spike1 - spike2)

counts, bins, patches = plt.hist(spike_diffs, bins=histo_bins)
plt.xlabel('Time lag (s)')
plt.ylabel('Count')
```

```
Text(0, 0.5, 'Count')
```



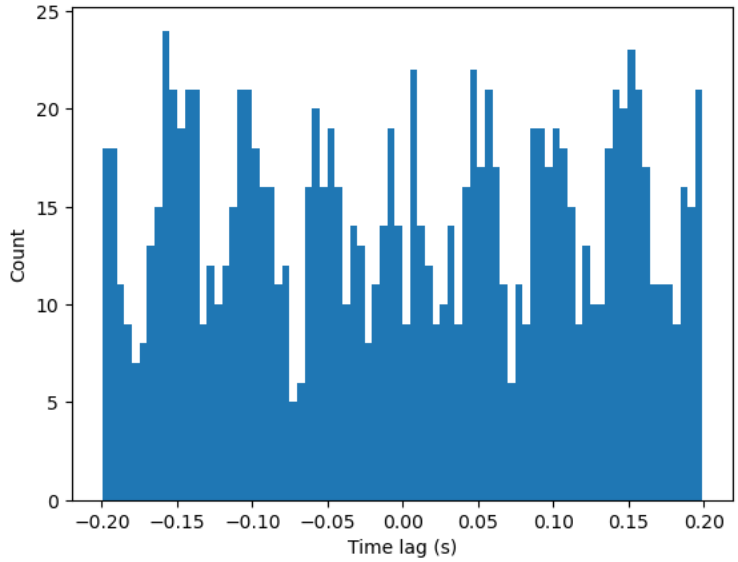
Here, we can see some rhythmicity in the comb-like appearence of the histogram, reflecting repeated time differences where spikes are more likely to occur. However, there is also a clear decrease in spike counts as the time differences increase. This is an artifact resulting from computing the autocorrelation at the beginning and end of the spike train. When comparing the time differences of other spikes relative to the first spike, time differences can only be positive, and for the last spike, time differences can be negative. Similarly, only spikes at the very beginning or end of the spike train can have a time difference with another spike close to 1. As we compute using spikes closer to the center of the spike train, the window of possible spike times increases, resulting in a linear increase in overall spike counts as time differences move closer to 0.

This artifact is really only relevant when computing auto-correlations on a short spike train, when the time differences of interest can approach the length of the spike train. Therefore, when computing an overall autocorrelation over the course of a recording, this will not be a problem, but when computing an autocorrelation over the course of a second or two (e.g. to see whether a stimulus induced rhythmicity), this affect will need to be addressed.

To remove this effect, establish the maximal time lag of interest. Then, we know which spikes to exclude: spikes that occur within the maximal lag of the start and end of the spike train.

```
# Compute autocorrelation for only valid spikes

# spikes = [t.item() for t in spikes] #convert spiketrain into a list of spike times

binsize = .005
max_lag = .2
start_win, stop_win = times[0] + max_lag, times[-1] - max_lag
histo_bins = int((max_lag*2)/binsize)

spike_diffs = []

for i, spike1 in enumerate(spikes):
    if spike1>start_win and spike1<stop_win:
        for j, spike2 in enumerate(spikes):
                if not spike1==spike2:
                    spike_diff = spike1 - spike2
                    if np.abs(spike_diff)<max_lag:
                        spike_diffs.append(spike1 - spike2)

counts, bins, patches = plt.hist(spike_diffs, bins=histo_bins)
plt.xlabel('Time lag (s)')
plt.ylabel('Count')
```

```
Text(0, 0.5, 'Count')
```



Here, we see peaks at -50 and 50 ms, and every multiple of that time lag, indicating rhythmic spiking every 50ms or 20Hz.

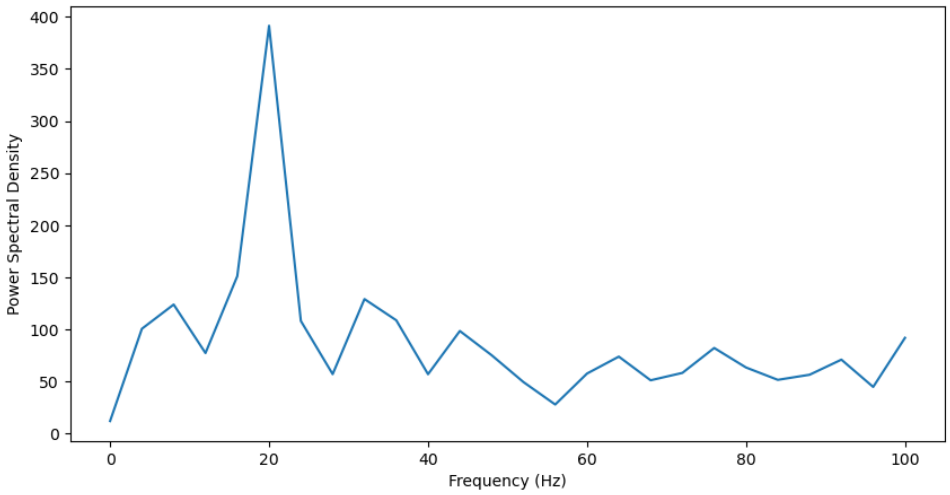
Power metrics of spike rate

Analyses applied to quantify LFP oscillatory power can also be applied to the spike rate of our spike train [3]. Here, the kernel size must be chosen carefully, as a kernel smoothing the spike train over a long period of time will smear high frequency rhythmic activity, while smoothing over short periods of time can lead to a noisy signal. Here, we'll analyze the power spectrum and time frequency power of the spike rate.

```
sampling_period = 1 * pq.ms
kernel_size = 5 * pq.ms
Fs = int(1000/float(sampling_period))
kernel_size = sampling_period
kerneled_spike_rate = elephant.statistics.instantaneous_rate(spiketrain, sampling_period=1
/ spike_sr * pq.s,

kernel=elephant.kernels.GaussianKernel(sigma=kernel_size))
kerneled_spike_rate = np.squeeze(np.asarray(kerneled_spike_rate))
```

```
f, Pxx = welch(kerneled_spike_rate, fs=fs, nperseg=250, noverlap=100)
f = f[np.where(f<=100)] #select only frequencies below 100Hz
Pxx = Pxx[np.where(f<=100)]
plt.figure(figsize=(10,5))
plt.plot(f, Pxx)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power Spectral Density')
```

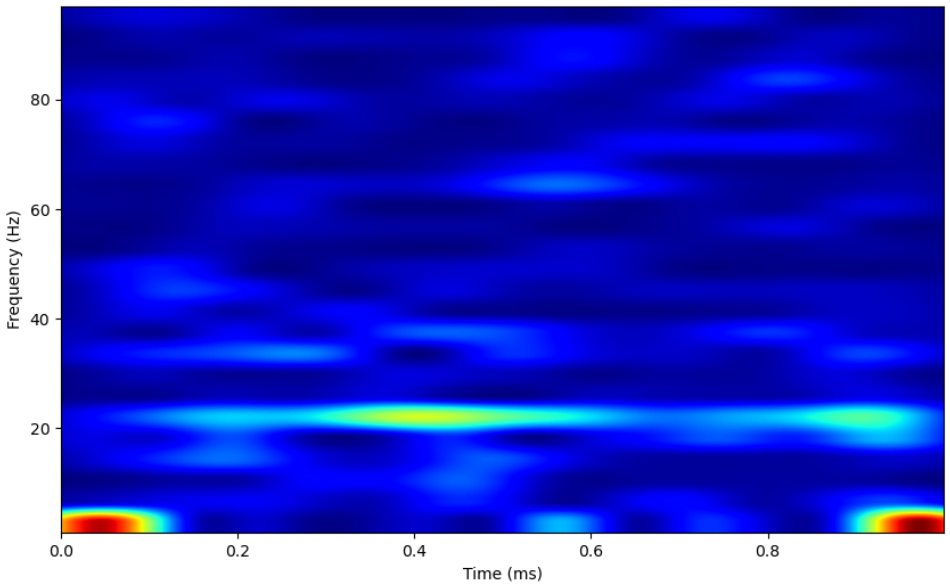
```
Text(0, 0.5, 'Power Spectral Density')
```




```
info = mne.create_info(ch_names=['1'], sfreq=Fs, ch_types=['eeg'])
epoch = np.empty((1, 1, len(kerneled_spike_rate)))
epoch[0,0,:] = kerneled_spike_rate #Format data into [epochs, channels, samples] format
epoch = mne.EpochsArray(epoch, info, verbose=False)
freqs = np.arange(1,100,4)
n_cycles = freqs / 2
tf_pow = mne.time_frequency.tfr_morlet(epoch, freqs=freqs, n_cycles=n_cycles,
return_itc=False)

tf_pow.data = np.squeeze(tf_pow.data)
plt.figure(figsize=(10,6))
plt.imshow(tf_pow.data, extent=[times[0], times[-1], tf_pow.freqs[0], tf_pow.freqs[-1]],
           aspect = 'auto', origin = 'lower', cmap='jet')
plt.xlabel('Time (ms)')
plt.ylabel('Frequency (Hz)')
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

```
Text(0, 0.5, 'Frequency (Hz)')
```



The power spectrum displays a strong peak at 20Hz, and the time-frequency representation shows some periods where 20Hz power is high. Overall, these methods are usually more effective when applied to LFP data, as the binary nature of neural spiking activity often results in noisy data. Still, these analyses can be useful in the absence of lfp data, or to analyze oscillatory activity individually in neurons.

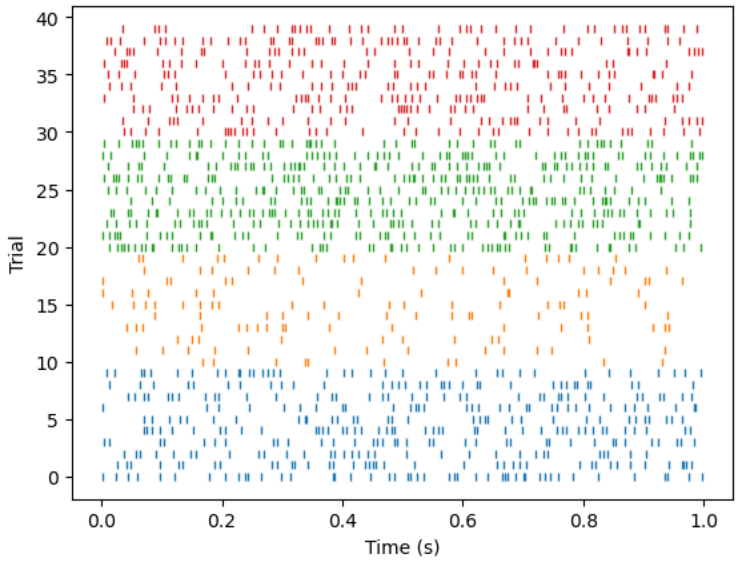
Stimulus encoding

A common step in electrophsiological experiments is to assess whether neural spiking activity can reflect stimulus information. To do so, we can start by plotting how individual neurons respond to different stimuli.

Let's say we have a neuron responding to 4 stimuli, where the neuron's response depends on the stimuli. Let's make a raster plot, where different stimuli are plotted using different colors.

```
n_stim = 4
n_units = 1

colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

# Make a matrix of stimulus response rates
response_rates = np.zeros((n_units, n_stim))
max_rate = 60
for u in range(0,n_units):
    for s in range(0,n_stim):
        response_rates[u, s] = max_rate*np.random.random([1])[0]

# Simulate trials for each neuron and stimulus, and count spikes over a second period
n_trials = 10
matrix_defined = False
trial_count = 0
for s in range(0,n_stim):
    response_counts = np.zeros((n_trials, n_units))
    for trial in range(0, n_trials):
        for u in range(0,n_units):
            spiketrain =
elephant.spike_train_generation.StationaryPoissonProcess(rate=response_rates[u, s]*pq.Hz,

refractory_period=3*pq.ms).generate_spiketrain()
            spikes = np.squeeze(spiketrain)
            plt.plot(spikes, trial_count*np.ones_like(spikes), '|', markersize=4,
color=colors[s])
            trial_count+=1

plt.xlabel('Time (s)')
plt.ylabel('Trial')
```

```
Text(0, 0.5, 'Trial')
```



The neuron's response rate should reflect the stimuli.

Now, what if we have 10 neurons, each with independent stimulus preferences. We'd like to be able to assess how these neurons could encode for stimuli as a population. However, we can't effectively visualize the activity of 10 neurons over 10 trials and 4 stimuli. Instead, we can use principle component analysis (PCA) to reduce the activity of the population of neurons into 2 or 3 components.

Principle Component Analysis

PCA reduces some features into a smaller number of components. In neuroscience, PCA is used to reduce the activity from some number of neurons into 2 or 3 components so that the data can be easily summarized and visualized. PCA works by iteratively finding the vector such that the projection of the data onto this vector best captures the variance in the data. The following vectors, or components, are then identified to capture the remaining variance in the data while being orthogonal to the previous components. Mathmatically, the fist step of PCA analysis is to compute the covariance matrix of the data. The components are then identified as the eigenvectors of the covariance matrix.

In interpreting the results of a PCA analysis, there are a few important points to remember. First, principle components are linear combinations of the initial variables: the actual output of a PCA decomposition is this weighting matrix, which must be multiplied with the original data to transform the data into PCA-space. Additionally, components are linearly uncorrelated with each other. Finally, since PCA iteratively captures as much variance as possible with each component, each subsequent component will capture less variance than the previous component [4].

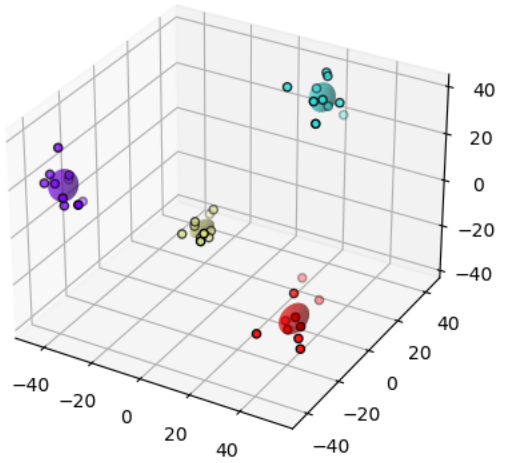
In sklearn's PCA function, the input matrix needs to be formatted such that the first dimension corresponds to each sample, and the second corresponds to each feature (here, neuron). We want to input all of our trials, regardless of the stimulus, into sklearn (but be able to organize by stimulus later), so here we'll stack trials from different stimuli along the first dimension, creating a matrix that is (n_stim x n_trials) by n_units.

```
n_stim = 4
n_units = 10

# Make a matrix of stimulus response rates
response_rates = np.zeros((n_units, n_stim))
max_rate = 60
for u in range(0,n_units):
    for s in range(0,n_stim):
        response_rates[u, s] = max_rate*np.random.random([1])[0]

# Simulate trials for each neuron and stimulus, and count spikes over a second period
n_trials = 10
matrix_defined = False
for s in range(0,n_stim):
    response_counts = np.zeros((n_trials, n_units))
    for trial in range(0, n_trials):
        for u in range(0,n_units):
            spiketrain =
elephant.spike_train_generation.StationaryPoissonProcess(rate=response_rates[u, s]*pq.Hz,

refractory_period=3*pq.ms).generate_spiketrain()
            response_counts[trial, u] = len(spiketrain)

    if matrix_defined:
        feature_matrix = np.concatenate((feature_matrix, response_counts), axis=0)
    else:
        feature_matrix = response_counts
        matrix_defined = True

feature_matrix.shape
```

```
(40, 10)
```

We can then perform PCA decomposition on this matrix, and plot the newly reduced dataset by which stimuli the trials came from.

```
#Decompose data using PCA
pca = PCA(n_components=3)

pca.fit(feature_matrix)
X = pca.transform(feature_matrix)

#Plot PCA
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

colors = plt.cm.rainbow(np.linspace(0, 1, n_stim))
for stim in range(0, n_stim):
    color = colors[stim]
    start = stim*n_trials
    stop = (stim+1)*n_trials
    ax.scatter(X[start:stop, 0], X[start:stop, 1], X[start:stop, 2], color=color,
edgecolor='k')

    pc1_mean, pc2_mean, pc3_mean = np.mean(X[start:stop, 0]), np.mean(X[start:stop, 1]),
np.mean(X[start:stop, 2])
    pc1_sem, pc2_sem, pc3_sem = np.std(X[start:stop, 0]), np.std(X[start:stop, 1]),
np.std(X[start:stop, 2])

    phi = np.linspace(0, 2 * np.pi, 256).reshape(256, 1)  # the angle of the projection in
the xy-plane
    theta = np.linspace(0, np.pi, 256).reshape(-1,256)  # the angle from the polar axis,
ie the polar angle
    x = pc1_sem * np.sin(theta) * np.cos(phi) + pc1_mean
    y = pc2_sem * np.sin(theta) * np.sin(phi) + pc2_mean
    z = pc3_sem * np.cos(theta) + pc3_mean
    ax.plot_surface(x, y, z, color=color, alpha=0.5,
                    linewidth=0)
```



The stimuli are clearly separated in PCA space.

References

1. Denker M, Yegenoglu A, Grün S (2018) Collaborative HPC-enabled workflows on the HBP Collaboratory using the Elephant framework. Neuroinformatics 2018, P19. doi:10.12751/incf.ni2018.0019
2. Sanderson, A. C. (1980). Adaptive filtering of neuronal spike train data. IEEE Transactions on Bio-Medical Engineering, 27(5), 271–274. https://doi.org/10.1109/TBME.1980.326633
3. Burton, S. D., & Urban, N. N. (2021). Cell and circuit origins of fast network oscillations in the mammalian main olfactory bulb. ELife, 10. https://doi.org/10.7554/ELIFE.74213
4. Hastie, T. et. all. (2009). Springer Series in Statistics The Elements of Statistical Learning. In The Mathematical Intelligencer (Vol. 27, Issue 2). http://www.springerlink.com/index/D7X7KX6772HQ2135.pdf

Spike Synchrony

Neural synchronization can facilitate information transfer and the binding of information. Many methods have been employed to quantify nerual synchrony, and no one technique has become widely viewed as the gold-standard. Here, I will illustrate several methods to quantify neural synchrony, specifically:

1. Cross correlation
2. Unitary event analysis
3. Spike-triggered population rate

```
import elephant
import quantities as pq
from neo.core import AnalogSignal
from neo import SpikeTrain
import numpy as np
import matplotlib.pyplot as plt
import quantities as pq
import random
import viziphant.unitary_event_analysis as vue
from scipy.stats import norm, binom
from seaborn import color_palette
```

First, let's simulate 2 neurons, synchronized to the same underlying oscillation. Here, we can model this by generating two spike trains from the same underlying rhythmic firing rate.

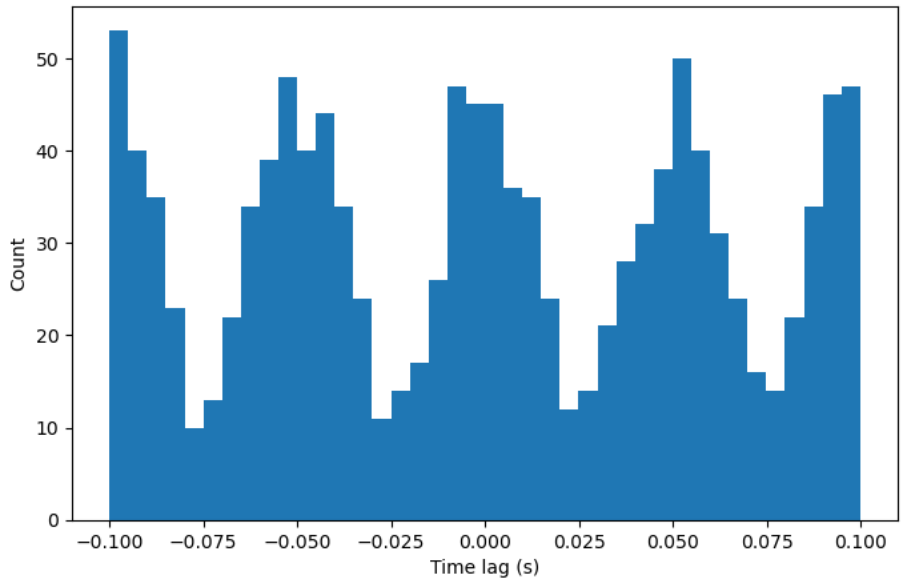
```
np.random.seed(1224)
fs = 1000
times = np.arange(0, 2, 1/fs)
freq = 20
mean_rate = 60
osc = np.sin(2 * np.pi * times[:] * freq)
osc_rate = mean_rate * osc
osc_rate += mean_rate #make sure rate is always positive
sim_spike_rate = AnalogSignal(np.expand_dims(osc_rate, 1), units='Hz',
sampling_rate=1000*pq.Hz)
st1 = elephant.spike_train_generation.inhomogeneous_poisson_process(rate=sim_spike_rate,
refractory_period=3*pq.ms)
st2 = elephant.spike_train_generation.inhomogeneous_poisson_process(rate=sim_spike_rate,
refractory_period=3*pq.ms)
```

Cross correlation

Generating a cross correleogram is a good first step for assessing possible synchrony between neurons. A cross correlelogram shows the approximate timescale of this synchrony, as well as the rhythm that gives rise to the synchrony. Additionally, a cross correlation is relatively straightforward and common to compute. For more details, see the section on **autocorrelation** in *spiketrain_analyses*. This is the same analysis, but instead of comparing a spike train with itself, here we're comparing two spike trains.

```
spikes1 = np.squeeze(st1)
spikes2 = np.squeeze(st2)

binsize = .005
max_lag = .1
start_win, stop_win = times[0] + max_lag, times[-1] - max_lag
histo_bins = int((max_lag*2)/binsize)

spike_diffs = []

for i, spike1 in enumerate(spikes1):
    if spike1>start_win and spike1<stop_win:
        for j, spike2 in enumerate(spikes2):
                spike_diff = spike1 - spike2
                if np.abs(spike_diff)<max_lag:
                    spike_diffs.append(spike1 - spike2)

plt.figure(figsize=(8,5))
counts, bins, patches = plt.hist(spike_diffs, bins=histo_bins)
plt.xlabel('Time lag (s)')
plt.ylabel('Count')
```

```
Text(0, 0.5, 'Count')
```



The bin at time lag = 0 can be used to assess synchronization. Here, the spike count at 0 is greater than the overall mean, suggesting these neurons are synchronous.

Unitary event analysis

Unitary event analysis essentially computes the number of synchronous spikes [1]. Because some number of synchronous spikes are expected by chance, this number is then compared to an 'expected' number of spikes.

```
# Compute number of synchronous spikes
binned_st1 = elephant.conversion.BinnedSpikeTrain(st1, bin_size=5*pq.ms) #Binarize spike
train
binned_st2 = elephant.conversion.BinnedSpikeTrain(st2, bin_size=5*pq.ms) #Binarize spike
train

sync_events = np.logical_and(binned_st1.to_bool_array(),binned_st2.to_bool_array()) #Find
where spikes occurred in both spiketrains
n_events = np.sum(sync_events)
n_bins = sync_events.shape[1]
print('Number of bins: '+ str(n_bins))
print('Number of synchronous events: ' + str(n_events))
```

```
Number of bins: 400
Number of synchronous events: 43
```

Now, let's compare this number with how much we would expect by chance. There are a few ways to compute this. Analytically, we have two spike trains with a mean rate of 60Hz. Therefore, each neuron spikes .3 times per 5ms, and for any 5ms bin, the probability of both neurons spiking is .09. We can calculate the probability that we observe 43 synchronous events or greater using a binomial distribution.

```
# Analytic
p = binom.sf(n_events, n_bins, .09)
print('Analytic probability: '+ str(p))
```

```
Analytic probability: 0.09754615335115961
```

Though simple and efficient, this method works only when the firing rate of neurons is stationary over time. However, the firing rate of neurons can change over time, in ways that are correlated with one another that may be spurious to this analysis. This can occur with the onset of a stimulus, or a neuromodulator.

To deal with non-stationarities, there are several methods to create surrogate spike trains, which the actual number of synchronous events can be compared with [2]. Surrogates are commonly created by dithering (or jittering) a spike train by randomly moving each spike by some time within some small time frame. Alternatively, surrogates can be created by randomly shifting spike trains against each other, conserving the original structure of each spike train but requiring ultimately large time shifts. When analyzing synchrony evoked by a stimulus, spike trains can be shifted across trials, such that the number of synchronous events can be compared with the number of synchronous events found when comparing neurons recorded in different trials. However, this analysis assumes stationarity can be assumed across trials, which is generally unlikely.

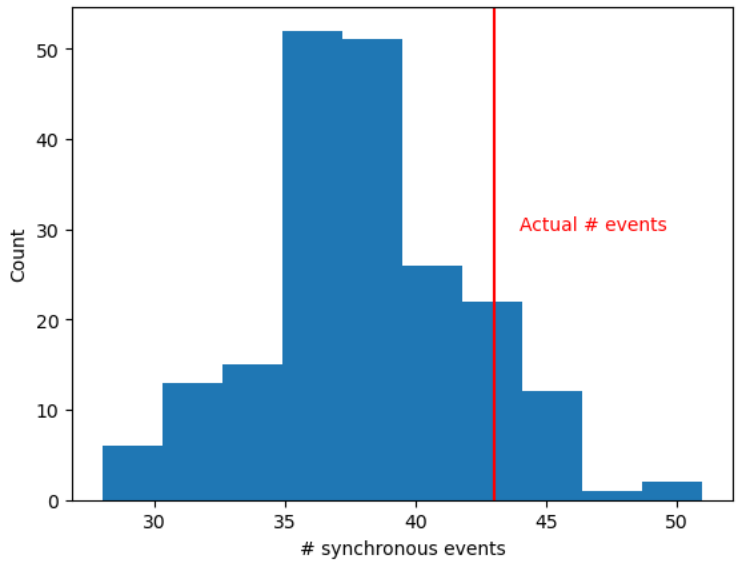
Here, let's dither to create surrogate spike trains, and compare the actual number of synchronous events with this distribution.

```
# Compare with surrogate spike trains
np.random.seed(2022)
surr_evts = []
surr_sts = elephant.spike_train_surrogates.dither_spikes(st2, 10*pq.ms, n_surrogates=200)
for surr_st in surr_sts:
    binned_surr_st = elephant.conversion.BinnedSpikeTrain(surr_st, bin_size=5*pq.ms)
#Binarize spike train
    sync_events =
np.logical_and(binned_st1.to_bool_array(),binned_surr_st.to_bool_array()) #Find where
spikes occurred in both spiketrains
    surr_evts.append(np.sum(sync_events))
```

```
C:\Users\Michael\anaconda3\envs\tutorials\lib\site-packages\quantities\quantity.py:337:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a
list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is
deprecated. If you meant to do this, you must specify 'dtype=object' when creating the
ndarray.
  return np.multiply(other, self)
```

```
plt.hist(surr_evts)
plt.axvline(n_events, color='red')
plt.xlabel('# synchronous events')
plt.ylabel('Count')
plt.text(n_events+1, 30, 'Actual # events', color='red')

z_score = (n_events - np.mean(surr_evts)) / np.std(surr_evts)

dith_p = norm.sf(z_score)
print('Dithered probability: '+ str(dith_p))
```

```
Dithered probability: 0.10832211715896994
```



Now, let's simulate two spike trains that only become synchronous after 1s.

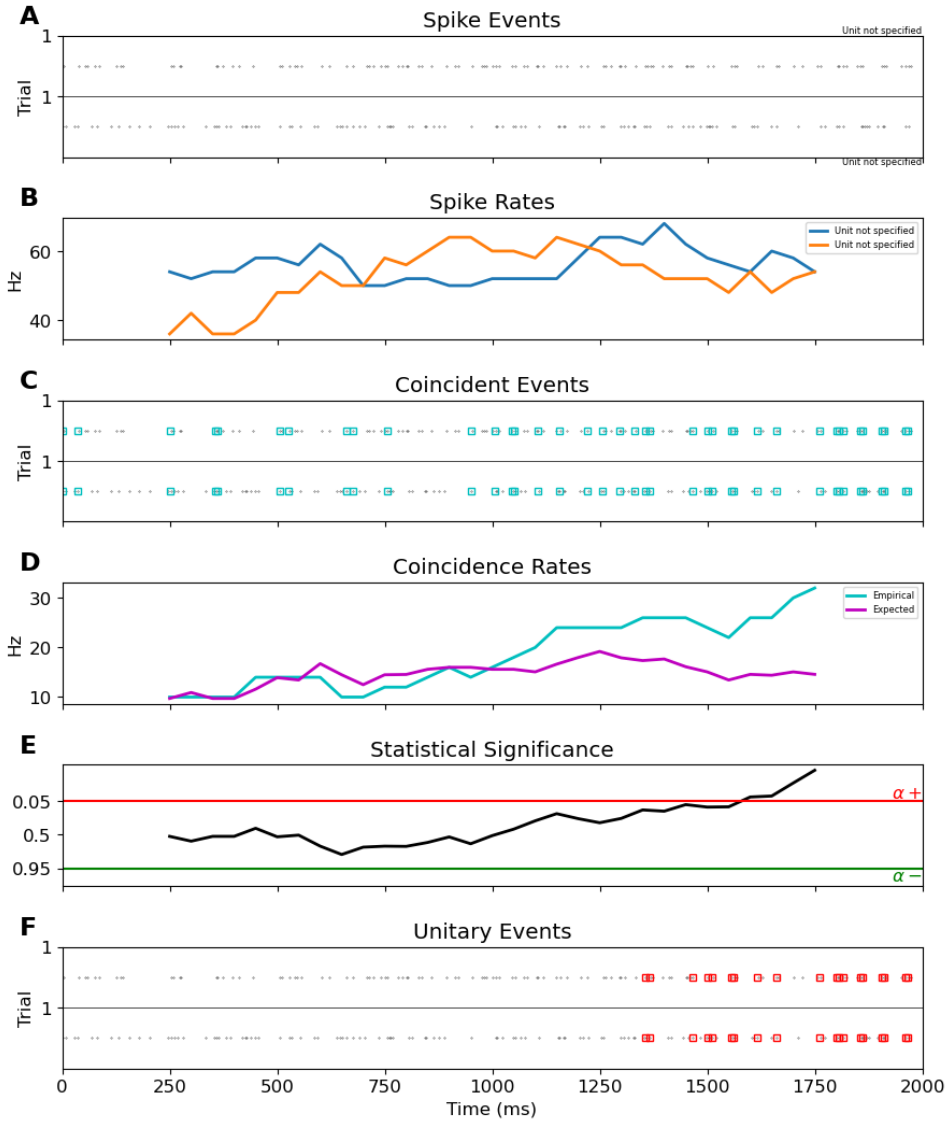
```
np.random.seed(2022)
fs = 1000
times = np.arange(0, 2, 1/fs)
freq = 20
mean_rate = 60
osc = np.sin(2 * np.pi * times[:] * freq)
osc_rate = np.zeros(len(times))
osc_rate[1000:] = mean_rate * osc[:1000]
osc_rate += mean_rate #make sure rate is always positive
sim_spike_rate = AnalogSignal(np.expand_dims(osc_rate, 1), units='Hz',
sampling_rate=1000*pq.Hz)
st1 = elephant.spike_train_generation.inhomogeneous_poisson_process(rate=sim_spike_rate,
refractory_period=3*pq.ms)
st2 = elephant.spike_train_generation.inhomogeneous_poisson_process(rate=sim_spike_rate,
refractory_period=3*pq.ms)
```

We can calculate how synchrony changes over time using elephant's jointJ_window_analysis function, and plot it using the plot_ue function from viziphant (elephant's companion visualization package).

```
spiketrains = [[st1, st2]]
random.seed(2022)
UEs = elephant.unitary_event_analysis.jointJ_window_analysis(spiketrains,
bin_size=5*pq.ms, winsize=500*pq.ms, winstep=50*pq.ms, pattern_hash=[3])
```

```
vue.plot_ue(spiketrains, UEs, significance_level=0.05)
```

```
FigureUE(axes_spike_events=<AxesSubplot: title={'center': 'Spike Events'},
ylabel='Trial'>, axes_spike_rates=<AxesSubplot: title={'center': 'Spike Rates'},
ylabel='Hz'>, axes_coincident_events=<AxesSubplot: title={'center': 'Coincident Events'},
ylabel='Trial'>, axes_coincidence_rates=<AxesSubplot: title={'center': 'Coincidence
Rates'}, ylabel='Hz'>, axes_significance=<AxesSubplot: title={'center': 'Statistical
Significance'}>, axes_unitary_events=<AxesSubplot: title={'center': 'Unitary Events'},
xlabel='Time (ms)', ylabel='Trial'>)
```



Elephant's visualization shows many coincident events. Additionally, the actual coincidence rate is generally higher than the expected, as calculated analytically. However, the windows only reach significance around 1s, and only coincident events in these windows are marked as unitary events.

Spike-triggered population rate

Alternatively, spike-spike synchrony can be computed as the spike-triggered population rate (stPR), or the firing rate of other neurons when one neuron fires [3]. This method has several advantages. First, the method can be used to test the synchrony between one neuron and a population of neurons. Therefore, when analyzing the general synchrony of each neuron, rather than neuron synchrony between individual neurons, this method can be much more efficient. Additionally, as demonstrated later, this method does not require the generation of surrogate neural spiketrains to control for a non-stationarity spikerate. As a result, this method can be much more computationally efficient in analyzing the neural synchrony between a neuron and a population of neurons. Finally, this method provides a synchrony value for each spike, rather than for a given window of time, allowing for finer time resolution in analyzing synchrony.

To illustrate this method, we'll start off by simulating a neuron, and a neural population, each firing in a synchronized rhythm.

```
#Simulate a neuron, and a neural population
np.random.seed(2022)
fs = 1000
times = np.arange(0, 2, 1/fs)
freq = 50
mean_rate = 60
osc = np.sin(2 * np.pi * times[:] * freq)
osc_rate = mean_rate * osc
osc_rate += mean_rate #make sure rate is always positiv
sim_spike_rate = AnalogSignal(np.expand_dims(osc_rate, 1), units='Hz',
sampling_rate=1000*pq.Hz)
st2 = elephant.spike_train_generation.inhomogeneous_poisson_process(rate=sim_spike_rate,
refractory_period=3*pq.ms)

n_pop_neurons = 10
st_pop = np.empty((0))
for n in range(0, n_pop_neurons):
    st =
elephant.spike_train_generation.inhomogeneous_poisson_process(rate=sim_spike_rate,
refractory_period=3*pq.ms)
    st = np.asarray(st)
    st_pop = np.concatenate((st_pop, np.squeeze(st)))

st_pop = SpikeTrain(np.squeeze(st_pop), units=pq.s, t_start=0 * pq.s, t_stop=(2 * pq.s))
```

Then, calculate the stPR, the mean firing rate of the population when neuron 2 fires.

```
#Calculate stPR (uncorrected)
spike_sr = 1000
sigma=5*pq.ms
kernel = elephant.kernels.GaussianKernel(sigma=sigma)
rate = elephant.statistics.instantaneous_rate(st_pop, sampling_period=1/spike_sr * pq.s,
                                              kernel=kernel, center_kernel=True)
rate = rate / n_pop_neurons
stPRs = []
spikes2 = np.squeeze(st2)
for spike in spikes2:
    stPRs.append(float(rate[int(round(spike * spike_sr, 4))]))

mean_stPR = np.mean(stPRs)
print('Mean firing rate: ' + str(np.mean(rate)))
print('Mean stPR: ' + str(mean_stPR*pq.Hz))
```

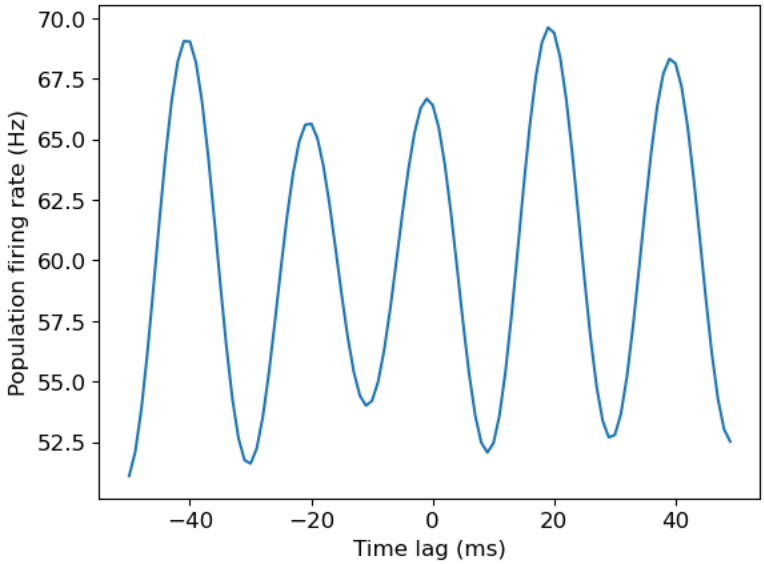
```
Mean firing rate: 59.576873252977244 Hz
Mean stPR: 66.67997370011862 Hz
```

The spike-triggered firing rate is greater than the overall mean firing rate, meaning neuron 2 is more likely to fire around when neurons in the population are firing relatively more.

Additionally, for each spike, we can get the firing rate in a surrounding window of time. This will allow us to see at what timescale the firing rate of neuron 1 increases when neuron 2 fires.

```
spike_sr = 1000
window = 100
sigma=5*pq.ms
kernel = elephant.kernels.GaussianKernel(sigma=sigma)
rate = elephant.statistics.instantaneous_rate(st_pop, sampling_period=1/spike_sr * pq.s,
                                              kernel=kernel, center_kernel=True)
rate = rate / n_pop_neurons
rate = np.asarray(rate)
stPR_segs = []
spikes2 = np.squeeze(st2)
for spike in spikes2:
    spike = float(spike)
    if spike > window/spike_sr and times[-1] - spike > window/spike_sr: #If window
entirely within times analyzed
        rate_seg = rate[int(round(spike*spike_sr))-
int(window/2):int(round(spike*spike_sr))+int(window/2)]
        stPR_segs.append(rate_seg)

stPRs_arr = np.asarray(stPR_segs)
stPRs_arr = np.squeeze(stPRs_arr)

plt.plot(np.arange(-window/2, window/2, 1), np.mean(stPRs_arr,0))
plt.xlabel('Time lag (ms)')
plt.ylabel('Population firing rate (Hz)')
```

```
Text(0, 0.5, 'Population firing rate (Hz)')
```



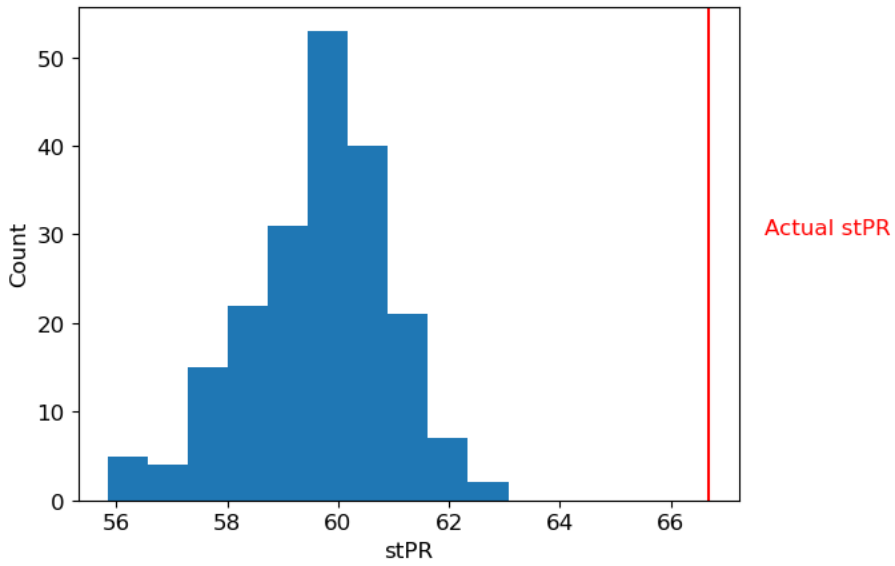
We have essentially created a cross correlation figure, using the firing rate of unit 1.

However, this method is biased by the mean firing rate of the population: if neurons fire more in general, the stPR will be higher. Thus, just as with unitary event analysis, we need to find a method to control for the overall firing rate of the population.

In the original paper for this method, the authors did so via a shuffling method. Though their exact method does not suit our data, we can implement something similar by comparing with surrogate spike trains as we did before.

```
#Surrogate vs. real stPR
np.random.seed(2022)
surr_stPRs = []
surr_sts = elephant.spike_train_surrogates.dither_spikes(st2, 10*pq.ms, n_surrogates=200)
for surr_st in surr_sts:
    stPRs = []
    surr_spikes = np.squeeze(surr_st)
    for spike in surr_spikes:
        stPRs.append(float(rate[int(round(spike * spike_sr, 4))]))
    surr_stPRs.append(np.mean(stPRs))

plt.hist(surr_stPRs)
plt.axvline(mean_stPR, color='red')
plt.xlabel('stPR')
plt.ylabel('Count')
plt.text(mean_stPR+1, 30, 'Actual stPR', color='red')

z_score = (mean_stPR - np.mean(surr_stPRs)) / np.std(surr_stPRs)

dith_p = norm.sf(z_score)
print('Dithered probability: '+ str(dith_p))
```

```
Dithered probability: 3.5099805001481054e-08
```

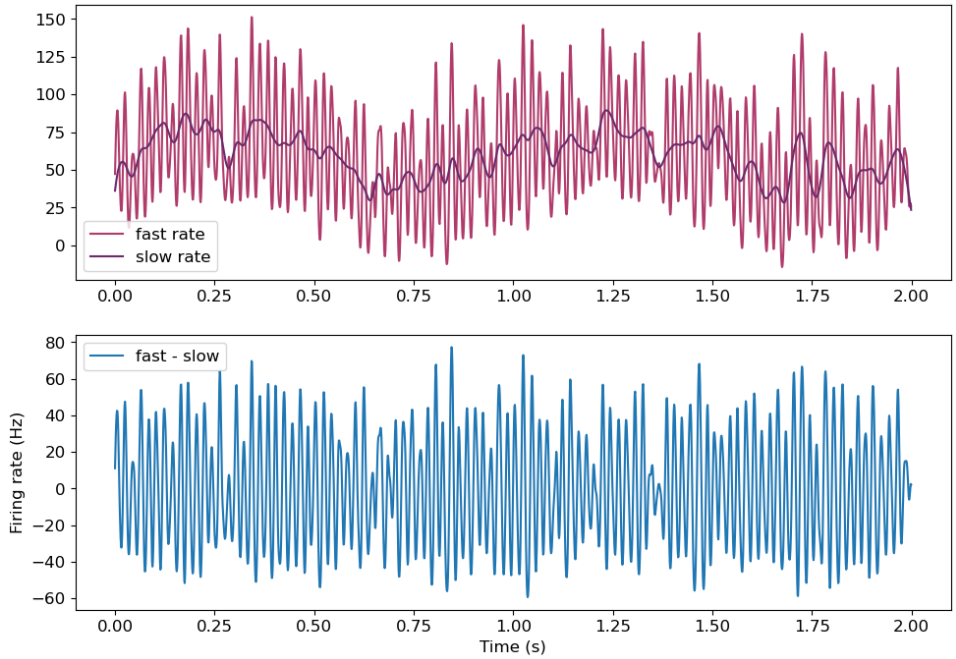


Alternatively, we can control for the overall firing rate of the population (and its nonstationarities) in our measurement of the firing rate itself. We can isolate the fine-timescale deviations from the mean firing rate by substracting out the longer-timescale estimate of firing rate. Specifically, this method measures firing rate on a fine time-scale using a narrow kernel, and subtracts the firing rate measured on a slow timescale from this firing rate.

First, let's isolate the fine-timescale fluctuations of the population by substracting out a longer-timescale estimate of firing rate. We'll add a simulation of a slow fluctuation in firing rate, to illustrate how we can remove it with this analysis.

```
#Isolate fine-timescale firing rate fluctuations
fast_kernel_sigma = 3*pq.ms
slow_kernel_sigma = 11*pq.ms
fastk = elephant.kernels.GaussianKernel(sigma=fast_kernel_sigma)
slowk = elephant.kernels.GaussianKernel(sigma=slow_kernel_sigma)

fast_rate = np.asarray(elephant.statistics.instantaneous_rate(st_pop,
sampling_period=1/spike_sr * pq.s,
                                                    kernel=fastk, center_kernel=True))
slow_rate = np.asarray(elephant.statistics.instantaneous_rate(st_pop,
sampling_period=1/spike_sr * pq.s,
                                                    kernel=slowk, center_kernel=True))
fast_rate /= n_pop_neurons
slow_rate /= n_pop_neurons
fast_rate = np.squeeze(fast_rate)
slow_rate = np.squeeze(slow_rate)
freq=1
osc = np.sin(2 * np.pi * times[:] * freq)
fast_rate += np.squeeze(osc*20) #Add general drift to the firing rates
slow_rate += np.squeeze(osc*20) #Add general drift to the firing rates
diff_rate = fast_rate - slow_rate
colors = color_palette("flare")
plt.figure(figsize=(11.5, 8))
plt.subplot(2,1,1)
plt.plot(times, fast_rate, color=colors[3], label='fast rate')
plt.plot(times, slow_rate, color=colors[-1], label='slow rate')
plt.legend()
plt.subplot(2,1,2)
plt.plot(times, diff_rate)
plt.legend(['fast - slow'])
plt.xlabel('Time (s)')
plt.ylabel('Firing rate (Hz)')
```

```
Text(0, 0.5, 'Firing rate (Hz)')
```



The fine-timescale kernel firing rate tracks the high frequency oscillation, while the slow-timescale kernel does not. Still, the slow kernel tracks the non-stationary firing rate. Substracting this from the fast kernel removes the slower drift in our data. Now, firing rate has a mean of 0, and positive values indicate a relative increase in firing rate, while negative values indicate the reverse. Now, we can calculate the stPR on this difference.

```
stPRs = []
spikes2 = np.squeeze(st2)
for spike in spikes2:
    stPRs.append(float(diff_rate[int(round(spike * spike_sr, 4))]))

mean_stPR = np.mean(stPRs)
print('Mean firing rate: ' + str(np.mean(diff_rate)*pq.Hz))
print('Mean stPR: ' + str(mean_stPR*pq.Hz))
```

```
Mean firing rate: 0.18442787923700246 Hz
Mean stPR: 16.170752011037074 Hz
```

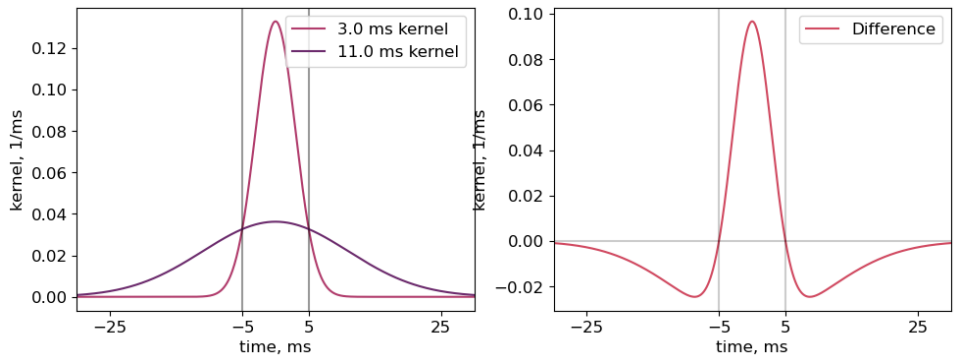
The stPR is much higher than 0, indicating the neuron is synchronous with the population. Notably, unlike the other methods, this analysis does not provide a significance value. Still, these values can be compared between neurons, or with a baseline.

Kernel design

We can get a more exact idea of what firing rate changes this timescale is estimating by plotting how each kernel estimates the firing rate of a single spike.

```
#Kernel design
fast_kernel_sigma = 3*pq.ms
slow_kernel_sigma = 11*pq.ms
spiketrain = SpikeTrain([-1, 0, 1], t_start=-1, t_stop=1, units='s')
time_array = np.linspace(-50, 50, num=1001) * pq.ms
kernel = elephant.kernels.GaussianKernel(sigma=fast_kernel_sigma)
fast_kernel_time = kernel(time_array)
kernel = elephant.kernels.GaussianKernel(sigma=slow_kernel_sigma)
slow_kernel_time = kernel(time_array)


colors = color_palette("flare")
plt.figure(figsize=(11.5, 4))
plt.subplot(1,2,1)
plt.plot(time_array, fast_kernel_time, color=colors[3])
plt.plot(time_array, slow_kernel_time, color=colors[-1])
plt.legend([str(fast_kernel_sigma)+' kernel', str(slow_kernel_sigma)+' kernel',
'Difference'])
plt.xlabel("time, ms")
plt.ylabel("kernel, 1/ms")
plt.xticks([-25, -5, 5, 25])
plt.axvline(-5, color='black', alpha=.4)
plt.axvline(5, color='black', alpha=.4)
plt.xlim(-30, 30)

plt.subplot(1,2,2)
plt.plot(time_array, fast_kernel_time-slow_kernel_time, color=colors[2])
plt.xlabel("time, ms")
plt.ylabel("kernel, 1/ms")
plt.axvline(-5, color='black', alpha=.2)
plt.axvline(5, color='black', alpha=.2)
plt.axhline(0, color='black', alpha=.2)
plt.xticks([-25, -5, 5, 25])
plt.xlim(-30, 30)
plt.legend(['Difference'])
```

```
<matplotlib.legend.Legend at 0x1d02aca3610>
```



The narrower kernel results in an estimatation of firing rate that only increases immediately surrounding the spike, but increases to a much higher degree. Specifically, using a 3ms and 11ms sigma for the fine and slow time-scale kernels, respectively, the fine-timescale firing rate is increased in the 10ms surrounding the spike, but decreased in the remaining 60ms. Therefore, this metric will be positive when the difference between two spike times is less than 5ms, or negative when the difference is between 5 and ~30ms.

Using this method, the timescales of the kernels should be carefully designed to reflect what the study would like to define as synchrony, and what general fluctuations should be removed.

Convolution of a cross correlation

Other studies have convolved the cross correlation histogram, rather than the spike trains themselves, to estimate synchrony. Specifically, after a cross correlelogram has been computed, slower covariations in firing rate are estimated by convolving the cross correlelogram with a gaussian kerenel. The significance of each bin can then be compared to the convolved correlelogram, asssuming a poisson distribution [4]. This method therefore allows for the estimation of significance without jittering. Additionally, this method can also be applied towards identifying peaks in the cross correlelogram at time lags other than 0, for example, to identify a monosynaptic connection between neurons from a peak at +3ms [5]. However, the time resolution of this method is limited, as a cross correlelogram must

be computed with a sufficent time window to include a reasonable estimate of the distribution at various time lags. This method has not been used by our lab, but the Buzsaki lab has published example code on Github.

References

1. Grün, S., Diesmann, M., Grammont, F., Riehle, A., & Aertsen, A. (1999). Detecting unitary events without discretization of time. Journal of neuroscience methods, 94(1), 67-79
2. Grün, S. (2009). Data-driven significance estimation for precise spike correlation. Journal of Neurophysiology, 101(3), 1126–1140. https://doi.org/10.1152/JN.00093.2008/ASSET/IMAGES/LARGE/Z9K0030993390009.JPEG
3. Okun, M., Steinmetz, N. A., Cossell, L., Iacaruso, M. F., Ko, H., Barthó, P., Moore, T., Hofer, S. B., Mrsic-Flogel, T. D., Carandini, M., & Harris, K. D. (2015). Diverse coupling of neurons to populations in sensory cortex. Nature, 521(7553), 511. https://doi.org/10.1038/NATURE14273
4. Stark, E., & Abeles, M. (2009). Unbiased estimation of precise temporal correlations between spike trains. Journal of Neuroscience Methods, 179(1), 90–100. https://doi.org/10.1016/J.JNEUMETH.2008.12.029
5. English, D. F., McKenzie, S., Evans, T., Kim, K., Yoon, E., & Buzsáki, G. (2017). Pyramidal cell-interneuron circuit architecture and dynamics in hippocampal networks. Neuron, 96(2), 505. https://doi.org/10.1016/J.NEURON.2017.09.033

By Michael Mariscal